



Università di Bologna

MASTER THESIS

*Convolutional Neural Networks for Efficient Object Detection on
Ultra Low-Power Platforms*

AUTHOR

Pedro José Pereira Vieito

THESIS ADVISORS

Prof. Dr. Luca Benini

Dr. Francesco Conti

SUPERVISOR

Dr. Francesc Moll Echeto

Master's Degree in Electronics Engineering

UNIVERSITAT POLITÈCNICA DE CATALUNYA

October 2017

ABSTRACT

At the University of Bologna, the Microelectronics Research Group has been working on smart data analytics on ultra-low-power sensors for the past few years. This smart analysis is in many cases based on convolutional neural networks as the fundamental tool to extract features and information out of various raw data streams. Applying these techniques on the acquisition device itself can help reducing data transfer and storage but requires neural network models with small memory footprint and a really constrained computation workload.

This work proposes a software architecture and advanced quantization techniques to obtain image classification models with high accuracy, small size and low memory footprint that can properly work on a low-power device. The design is specifically tailored to support the low-resolution environment available in the PULP platform, which includes a hardware convolution engine to efficiently compute convolution operations required by neural network models.

CONTENTS

1 INTRODUCTION	8
1.1 Description	8
1.2 Objectives.....	9
1.3 Content	9
2 CONVOLUTIONAL NEURAL NETWORKS	10
2.1 Neural Networks	10
2.2 Structure	10
2.3 Layers.....	11
2.3.1 Dense Layer	11
2.3.2 Convolutional Layer	12
2.3.3 Activation Layer.....	13
2.3.4 Pooling Layer	14
2.4 Training	14
2.4.1 Loss function	14
2.4.2 Optimizer.....	14
2.4.3 Gradient-based learning	15
2.4.4 Back-propagation.....	15
2.5 Image Recognition.....	15
2.6 Topologies.....	16
2.6.1 LeNet	16
2.6.2 AlexNet	17
2.6.3 GoogLeNet	17
2.6.4 VGG	17
2.6.5 SqueezeNet	17
2.7 Deep Learning Software	18
2.7.1 Caffe.....	18
2.7.2 Torch	18
2.7.3 Theano.....	18

2.7.4 TensorFlow	18
2.7.5 Keras	18
3 SOFTWARE ARCHITECTURE	19
3.1 Description	19
3.2 NeuralNetTool	20
3.2.1 Features	20
3.2.2 Command Line Interface	20
3.2.3 Dataset Loader	21
3.2.4 Execution Flow	21
3.2.5 Interoperability	23
3.3 NeuralNetModel	24
3.3.1 GenericNet	24
3.3.2 Models	25
3.4 NeuralNetCore	25
4 SQUEEZENET DESIGN	27
4.1 Introduction	27
4.2 Topology	27
4.2.1 Strategies	27
4.2.2 Fire Module	27
4.2.3 Architecture	29
4.3 Implementation	30
4.3.1 Model Tensor	30
4.3.2 Fire Module	30
4.3.3 SqueezeNet	31
4.3.4 Verification	32
4.4 Quantization	32
4.4.1 Simulated Quantization	33
4.5 Quantization Implementation	34
4.5.1 Quantclip Operator	34

4.5.2 QuantizedConv2D.....	35
4.5.3 Quantized Fire Module	38
4.5.4 QuantizedSqueezeNet	39
5 EXPERIMENTAL RESULTS	42
5.1 Training	42
5.1.1 Training Setup	42
5.1.2 Training Results.....	43
5.2 Model Data Analysis.....	44
5.2.1 Weigh Data Analysis.....	44
5.2.2 Convolution Output Analysis	46
5.3 Off-model Quantization	47
5.3.1 Quantclip Function.....	47
5.3.2 Kernel Clipping	48
5.3.3 Kernel Resolution	49
5.4 On-model Quantization	49
5.4.1 Kernel Quantization.....	49
5.4.2 Kernel + Input Quantization	50
5.4.3 Full Quantization.....	51
5.5 Dynamic Quantization	52
5.5.1 Introduction	52
5.5.2 Quantize-and-Lock.....	52
5.5.3 Quantize-and-Lock + Custom MSB	54
5.5.4 Quantize-while-Train	56
5.5.5 Long Duration Quantize-while-Train	58
6 CONCLUSIONS AND FUTURE DEVELOPMENT	60
6.1 Conclusions	60
6.2 Future Development	60
7 BIBLIOGRAPHY	61

1 INTRODUCTION

1.1 DESCRIPTION

At the University of Bologna, the Microelectronics Research Group has been working on smart data analysis on ultra-low-power sensors for the past few years along the entire technological stack, from the acquisition hardware to the software running on microcontrollers. This smart analysis is in many cases based on convolutional neural networks (CNN) as the fundamental tool to extract features and information out of various raw data streams.

The amount of data needed to be transferred from a low-power sensor to the final storage and interfacing device can be greatly reduced by the implementation of this technology. The main limitation of this approach is that the data processing in the low-power device must have an extremely small memory footprint and a really constrained computation workload. This is a big inconvenience when working with CNN, as they require a lot of mathematical operations to execute the model.

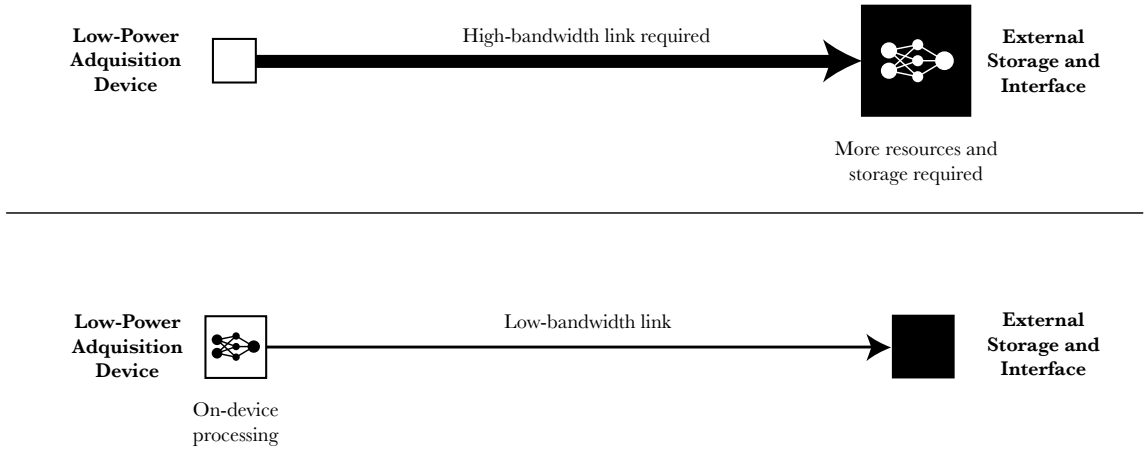


Figure 1: The on-device processing approach brings multiple advantages to an acquisition system.

The solution proposed in this work is to select a relatively small CNN model with a good object classification accuracy and adapt it to the hardware constraints of a low-power embedded device such as PULP (Parallel Ultra-Low-Power Platform), a hardware computing platform developed in conjunction by the University of Bologna and ETH Zurich Institute.

On this context, the most important feature of the PULP platform is the presence of a hardware convolution engine (HWCE) to efficiently compute convolution operations (Conti & Benini, 2015) while reducing the power consumption of the device during the model execution.

This hardware also imposes some constraints on the structure of the convolutional network, as the HWCE is designed to work with convolution filters of small resolution and size. Therefore, it is critical to ensure that the model is properly adapted to a low-resolution environment and works correctly on it.

The core of this work was the design of a CNN model with high accuracy, small size and low memory footprint that can properly work with low-resolution convolutions required by a low-power device.

1.2 OBJECTIVES

The main goals of this project were the following:

- Design a software system capable of training, evaluating and fine-tuning a convolutional neural network.
- Ensure this software is scalable from a server with a powerful GPU array to a low-power custom embedded device.
- Select and adapt an existing small and efficient CNN model for image classification.
- Replicate the accuracy of the existing model with our custom training software.
- Retrain the model from scratch, using our own full dataset to achieve similar accuracy to the one of the original model.
- Research, evaluate and apply different strategies to adapt a CNN model to a low-resolution environment similar to one available in a low-power platform.

1.3 CONTENT

This work comprises of a total of six chapters. After this brief introduction, the following chapter will present the concept of Convolutional Neural Networks, the process to train CNN models, how they are designed for image classification tasks and the libraries available to implement Deep Learning software.

The third chapter will explain the software architecture we implemented to train, evaluate, fine-tune and test various neural network models with different parameters and topologies.

In the fourth chapter, we will detail the selection and modification of a small neural network model. Also, the structure and topology of the model will be detailed.

The fundamental block of this work is the fifth chapter where we will introduce an extensive description of how we have minimized the model size and adapted it to work with low-resolution convolutions. We will point out the different techniques tried and the experimental results obtained in each case.

Finally, the last chapter will outline the conclusions drawn from the results of the project, examining and highlighting the most significant decisions. Possible lines of development will also be mentioned.

2 CONVOLUTIONAL NEURAL NETWORKS

2.1 NEURAL NETWORKS

An Artificial Neural Network is a complex computing system inspired in the biological brain structure found on nature. These systems are designed to be trained on a huge number of inputs with known outputs. During this training, the model will learn a pattern allowing it to predict the output of other additional inputs.

Usually this training requires large datasets that have been previously tagged with the interested output feature. During the training phase, the internal parameters of the neural network model are modified, enhancing the accuracy of the model in each iteration. This process must be repeated in batches (*epochs*) until the model achieves the desired accuracy.

An interesting feature of neural networks is that they are largely input agnostic before the training phase. Thus, one neural network model can be trained and adapted to use in a lot of different tasks. This approach allows neural networks to be implemented in a grand variety of technology areas like computer vision, speech recognition, machine translation, video games or social network filtering where the decision or classification task can be extremely difficult to define based on a limited set of rules or instructions.

2.2 STRUCTURE

The basic structure of any neural network consists on some inputs, some outputs and several hidden layers which connect the inputs and the outputs.

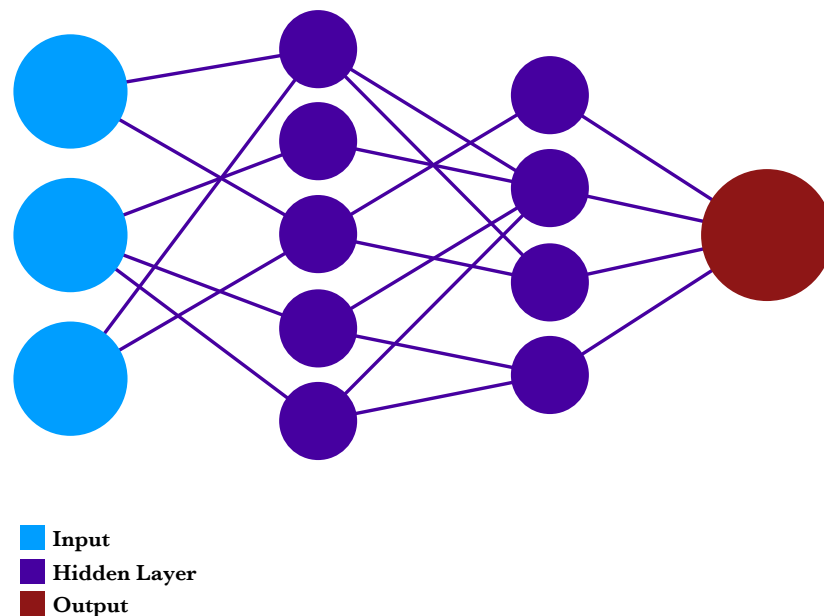


Figure 2: Schematic structure of a neural network.

These layers are connected through operations which include the trainable parameters that will be tuned during the learning phase. Once the model has been tuned, we can lock the internal parameters and use it for inference, predicting output features on additional input data.

2.3 LAYERS

Each connection between nodes inside the network is called a layer. Each layer executes some mathematical operation on its inputs and generates an output. There are a lot of different layer types and the most common ones will be outlined in this section.

2.3.1 Dense Layer

Dense layers (also known as fully connected layers) are highly trainable layers which connect each input value to all the outputs with a weighting parameter. Although they are the best way to connect any input size data to a specified output they require large number of trainable parameters making the model both big and extremely slow to train and execute.

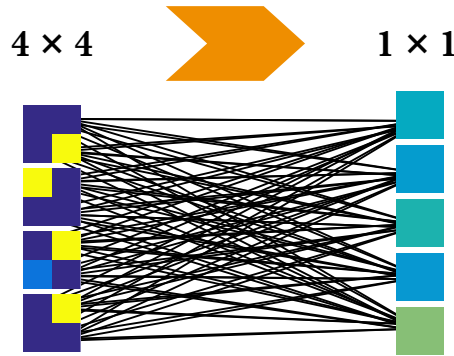


Figure 3: All the connections of a dense layer between four 2x2 images and five simple outputs. This figure shows how dense layers can have a very noticeable size even on small configurations

As all input values are connected to all output values, for n_i inputs of size s_i and n_o outputs of size s_o the model would include the following number of parameters:

$$d_{parameters} = n_i * s_i * n_o * s_o$$

Equation 1: Number of parameters on a dense layer based on the number and size of the input and output.

The final size of the weight data in the model can be obtained multiplying the number of parameters by the size of the storage data type (typically a *float* of 32 bits):

$$d_{size} = d_{parameters} * data_size$$

Equation 2: Storage size of the parameters of the dense layer.

Consequently, for example, a small dense layer connecting four 2x2 images and five outputs the model should include 320 parameters:

$$d_{parameters} = 4 * (4 * 4) * 5 * 1 = 320 \text{ parameters}$$

$$d_{size} = 320 \text{ connections} * 4 \text{ bytes} = 1280 \text{ bytes}$$

Equation 3: Example of the connections and parameter size of a small dense layer.

This example shows how the size of a dense layer can easily exponentially grow when the input nodes and outputs increases. Henceforth, dense layers are unfit for the design of small models

with low memory footprint and we will try to avoid any neural network topology that depends on them.

2.3.2 Convolutional Layer

Convolutional layers are the main building block of convolutional neural networks. They are formed by a convolution between the input data and an internal trainable weight matrix (also known as the convolution kernel or filter).

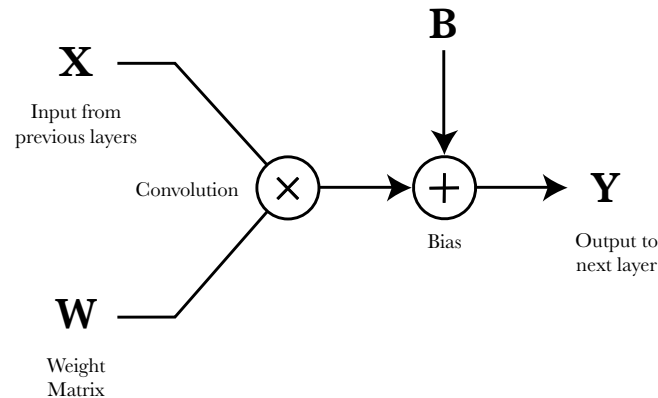


Figure 4: Structure of a convolutional layer; the weight matrix and the bias are its trainable parameters.

Usually after the convolution operation a constant is added to the result. This constant is called the convolution bias and is required to achieve good results during the training of non-normalized data.

Convolutional layers are great at extracting features from their input, so they are extremely effective in classification tasks like of object detection on images. Using various filters in parallel, convolutional layers can convert spatial information into the feature dimension, extracting each filter a determined classification feature.

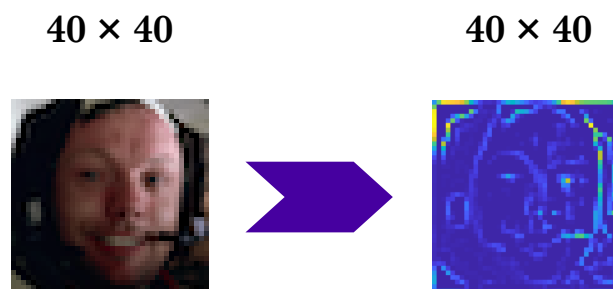


Figure 5: Convolution operation applied on an image which extracts some high frequency features.

In contrast to dense layers, convolutional layers store the weight or kernel matrix in a much smaller size, as they share the same kernel parameters for all the input pixels.

On a convolutional layer the number of filters, kernel size and convolution parameters determines the output size and the kernel values the effect of the filter applied.

2.3.3 Activation Layer

Activation layers are usually intercalated between other type of layers. They are designed to highlight important data values from their input. Historically, there have been three popular activation functions:

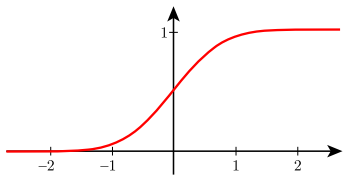
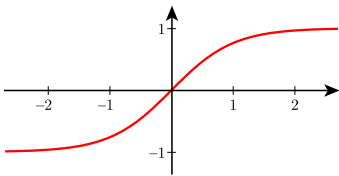
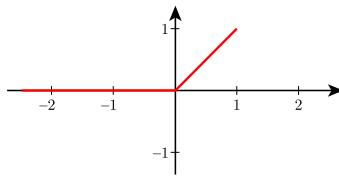
Sigmoid	Hyperbolic Tangent	Rectified Linear Unit
sigma	tanh	ReLU
$\frac{1}{1 + e^{-x}}$	$\tanh(x)$	$\max(0, x)$
		

Table 1: Main activation functions used on neural network models.

The most basic and early activation function is the sigmoid. It takes an input value and squashes it between 0 and 1. This means that large numbers become 1 and negative numbers become 0.

The next activation is the hyperbolic tangent function. Like the sigmoid, it saturates the input value, but this time with a zero-centered output, which is typically preferred while training the models.

ReLU has become the most popular activation function in the last years. It acts like a simple thresholder to zero when the value is negative and a pass-through when positive. It has two big advantages in comparison to the previous ones:

- It greatly accelerates the convergence of the model gradient descent (Krizhevsky, Sutskever, & Hinton, 2012), which speeds-up the training phase.
- Compared to other activation functions it can be implemented with an extremely simple mathematical operation, thresholding input values at zero.

The main defect of the ReLU activation function is that it is not robust against large gradients, which can make an activation point stale at a constant zero value. However, this is not typically a problem and ReLU has become the default activation function recommended for most neural networks.

Finally, it is important to note that there are a lot of variations of the ReLU activation function optimized for different use-cases like Leaky ReLU, softmax or ELU.

2.3.4 Pooling Layer

Pooling layers are used to reduce the number of data points on a given input. They are non-trainable layers as they simply combine adjacent data points down-sampling the input.

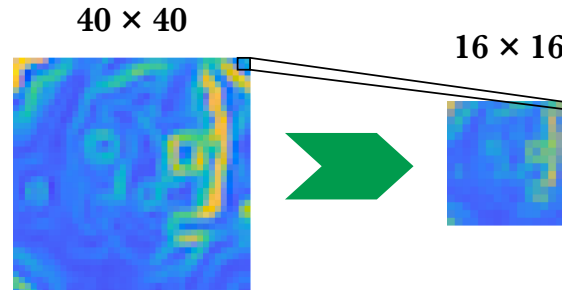


Figure 6: A pooling operation applied to a 40×40 image to output a 16×16 image.

Some convolutional neural networks try to avoid the use of pooling layers by modifying the data size directly during the convolutions, reducing the number of layers and enhancing the efficiency and speed of the model. This result can be achieved configuring each convolutional layer with a stride higher than one, which means that the convolution filter would not be applied to each input pixel, but instead would skip some of them, producing a smaller output.

2.4 TRAINING

Training is one of the fundamental steps required to achieve a useful neural network model. All deep learning algorithms must include the following pieces to be properly trained:

- A model with inputs, outputs, connections and trainable parameters.
- A tagged dataset to train and evaluate of the model.
- A cost function that can compute a statistical estimation related to the model accuracy.
- An optimization strategy to update the model parameters while increasing its accuracy.

2.4.1 Loss function

As the objective of the training is to maximize the accuracy of the model on a given task, there must be a method to compute the exactness of the model during the process. This is achieved with a cost function (or loss function) which represents a statistical estimation inversely related to the accuracy of the model.

The loss function is a fundamental piece of the learning phase as it provides the tool required to update the model on each iteration of the optimization.

2.4.2 Optimizer

The objective of an optimizer is to find the parameter values that minimizes the loss function. To achieve this, multiple algorithms are available:

- Random Search: try multiple random parameters and choose the one that resolves the minimum loss function value. This approach is really simple but will not achieve any

great results due to its random nature.

- **Random Local Search:** the model would start with some random parameters and then the optimizer would try to update them to obtain better results in the loss function. If a better result is found then it would continue updating the weight parameters in that direction, if not, it would try in another random direction.
- **Gradient-Based Learning:** this method is the one really used in the practice. It applies an algorithm similar to the random local search optimizer but it also computes the best direction (using the loss function gradient) on which to update the model parameters.

2.4.3 Gradient-based learning

The core operation of a gradient-based learning optimizers is defined in the following way, being w the previous iteration parameter matrix, w' the updated parameter matrix, δ the step size and $\nabla_w(L)$ the gradient of the loss function over the weight matrix:

$$w' = w - \delta * \nabla_w(L)$$

Equation 4: Formula to compute the updated weight matrix after each iteration following the gradient direction of the loss function with a defined step size.

This operation is executed in loop, with each iteration updating the model parameters towards the loss function local minimum. That point defines the parameters that produce a local maximum accuracy on the model.

Of course, detecting a local minimum in the loss function does not guarantee that the given point is the absolute minimum, so the optimizer has to deal with various strategies to find a valuable accuracy. These techniques involve modifying the step size of each update to change the path towards the maximum accuracy.

Gradient computation can be a hard task on complex models, as it requires to derive the function in each dimension, so in real Deep Learning software the implementation uses a technique called back-propagation, which is described in the next section.

2.4.4 Back-propagation

Back-propagation is a technique used to efficiently compute the loss function gradient. It separates the gradient calculation in parallelizable operations on each model node to propagate back the output error and compute each parameter deviation. Once those error values are computed the optimizer updates the parameters using the step factor previously defined.

2.5 IMAGE RECOGNITION

Convolutional neural networks are a great tool for image recognition and classification. Usually CNN models for image recognition have a common basic structure that we will describe in the following way:

- An input designed for images of a determined size, usually a square of around 200×200 pixels and with 3 color channels. Early CNN models, like LeNet, were designed for smaller input images, like 20×20 pixels with only one channel.
- Some convolutional layers, intercalated with ReLU activations to extract the features of the input image.

- Some pooling layers to reduce the resolution of the convolutional layers outputs.
- And finally, a fully connected layer to redistribute the result of the last convolutional layer over the outputs. Typically, the model has as many outputs as image categories, and each output represents the probability that the input image is included in that category. On binary classifier —like cat vs. dog or smiling face vs. not— only one output used, which represents the probability of the image being one of the cases.

The following graph shows an archetypal example of a CNN model for binary classification. Its input is a 40×40 image with three channels, it includes some convolutional, pooling and dense layers and finally, it outputs the probability of the image containing a smile.

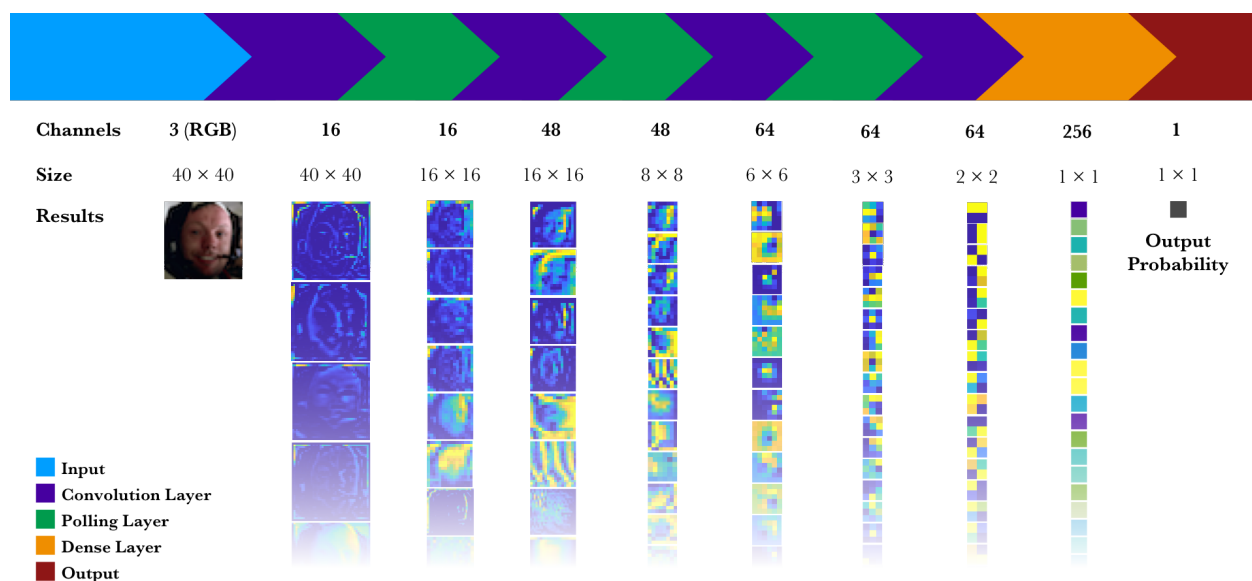


Figure 7: Structure and results of each layer of an image recognition convolutional neural network with only one output feature.

2.6 TOPOLOGIES

2.6.1 LeNet

LeNet (LeCun, Bottou, Bengio, & Haffner, 1998) was one of the first neural network models implemented in a commercial product. The design was developed in Bell Labs to allow machines to read hand-written digits. Prior to its design, image recognition software was implemented in two different pieces:

- A *Feature Extraction Module* which was manually designed to extract some interesting features of the input image like its frequency content or borders.
- A *Trainable Classifier Module* implemented with a model trained with preclassified data.

The main innovation of LeNet was that the full implementation could be trained with a preclassified dataset. Both the feature extractor and the classifier were unified in a trainable model. This made possible obtaining a better feature extractor that could be enhanced with the big datasets available.

The second innovation of LeNet was the introduction of convolutional layers in its neural network topology. Convolutional layers allowed for the first time to efficiently extract different image features in what the authors called feature maps.

2.6.2 AlexNet

AlexNet (Krizhevsky, Sutskever, & Hinton, 2012) is as CNN model developed in 2012 to achieve high accuracies in complex datasets like ImageNet (1000 categories). Using the ILSVRC-2010 dataset AlexNet achieved an accuracy of 62.5%.

The architecture of AlexNet includes five convolutional layers and three fully-connected layers. The big innovations of AlexNet were the following:

- The introduction of the ReLU non-linearity as an activation function instead of the classical *tanh* or *sigmoid* functions, which allowed it to converge quicker.
- Use of complex convolutional layers in a deep neural network model.

2.6.3 GoogLeNet

GoogLeNet (Szegedy, Liu, & Jia, 2014) is a convolutional network model designed by Google and winner of the ILSVRC-2014 contest. Its main innovation was the inclusion of a module named *Inception* designed to intensely reduce the number of parameters in the model (GoogLeNet used only 7% of the parameters required by AlexNet).

Moreover, GoogLeNet reduced the number of trainable parameters by replacing all dense layers in the model by convolutional and pooling layer pairs.

2.6.4 VGG

VGG (Simonyan & Zisserman, 2015) is a CNN model designed by the Department of Engineering Science of the University of Oxford. It was the runner-up of the ILSVRC-2014 competition.

Its main contribution was to show that the depth of a neural network (the number of layers) is a critical component to obtain a high accuracy model. Unfortunately, VGG is a really big network, with over 140 million parameters, which is not appropriate for low-power devices.

2.6.5 SqueezeNet

SqueezeNet (Iandola, Moskewicz, Ashraf, Han, & Keutzer, 2016) is a CNN model designed to have both small size and high accuracy. The first version of SqueezeNet achieved an accuracy of 57.5% on ImageNet, close to the one accomplished by AlexNet while requiring 50 times less parameters. Moreover, the total size of the SqueezeNet model is under 5 MB allowing for the first time to embed a complex CNN classifier on low-power devices with limited memory.

One of the main innovations of SqueezeNet was the introduction of the *Fire* module, which could efficiently extract features from the input images with few trainable parameters. Furthermore, to decrease the number of parameters, the model avoids any use of fully connected layers or big kernels on the convolutional layers.

All these features make SqueezeNet an outstanding model for the objective of this work. As explained in the following chapters we used the SqueezeNet topology as the base for our quantized model and its design and implementation is deeply described in the fourth chapter.

2.7 DEEP LEARNING SOFTWARE

There are multiple Deep Learning libraries available to ease the design, training and execution of convolutional neural network models. This software can be used for several applications like computer vision, machine translation or emotion detection. As running convolutional neural networks require a lot of mathematical operations, Deep Learning software implements low-level libraries which use any hardware acceleration available, like GPUs or the HWCE.

In this section, we briefly detail multiple Deep Learning software libraries available.

2.7.1 *Caffe*

Caffe (Jia, 2013) is one of the most popular and wide-spread deep learning frameworks available. It is written in C++ and includes bindings for different languages like C or Python. Its main benefits are its speed, performance and the huge quantity of models already available.

Caffe was created on the Artificial Intelligence Research Department of the University of Berkeley.

2.7.2 *Torch*

Torch (Collobert, 2011) is a scientific computing library with support for machine learning algorithms designed to use the hardware acceleration of the GPU implementing the CUDA API. It is written in C and Lua and its API is exposed through the scripting language LuaJIT.

2.7.3 *Theano*

Theano (Bergstra, 2010) was a low-level mathematical library written in Python. It was written at the LISA lab of the University of Montreal to support rapid development of efficient machine learning algorithms.

Though highly popular on the inception of Deep Learning software, Theano ceased its development after the introduction of competing libraries backed by strong industrial players, like TensorFlow.

2.7.4 *TensorFlow*

TensorFlow (Abadi, 2015) is a modern numerical computation library based on data flow graphs. It is written in C++ and CUDA and includes a Python API. TensorFlow works with computational graphs that are complex combinations of mathematical operations. They can be easily used to represent a neural network which makes TensorFlow extremely well fitted for Deep Learning applications.

TensorFlow was originally developed by researchers and engineers from Google Brain, the main team in charge of machine learning research at Google.

2.7.5 *Keras*

Keras (Chollet, 2015) is a high level Deep Learning library. It is fully written in Python and has an advanced API to implement neural network models and ease its training and execution. It was developed with a focus on enabling fast experimentation and prototyping.

Keras was designed to allow both Theano and TensorFlow as its mathematical backends so its code is extremely portable on a grand variety of devices and environments.

3 SOFTWARE ARCHITECTURE

3.1 DESCRIPTION

We selected Keras as the framework on which we would implement the software architecture of the project. The main advantages of Keras is that it has an extensive API, good documentation and community support. Moreover, there are multiple solutions available to import and export models from Keras, which will be required during the development of project.

The most important requirement of the software implementation is that it must run on a wide variety of environments with extremely different resources available. It must also be modular, so multiple models can be easily trained and tested.

To achieve these goals, the system was designed as three different modules:

- **NeuralNetTool:** the command line interface to evaluate, train, fine-tune and extract the state of a model.
- **NeuralNetModel:** the architecture that each model will have to implement to be compatible with NeuralNetTool. Using the same design, different models and topologies can be trained and tested with diverse parameters directly from the command line, speeding up the development and research.
- **NeuralNetCore:** the low-level module that must implement the hardware acceleration for the math operations needed to evaluate the neural network model on a low-power device.

NeuralNetTool is the interface designed to load the different models implemented using the NeuralNetModel architecture. Under the hood, NeuralNet uses Keras to load the model and execute its training and evaluation. Keras implements TensorFlow as its backend, which will process the low-level mathematical operations with the best hardware acceleration available on the system.

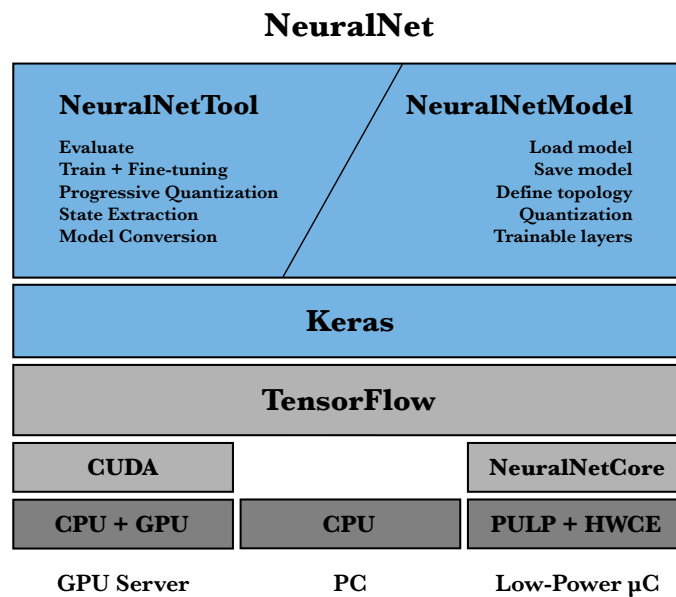


Figure 8: The NeuralNet system architecture.

3.2 NEURALNETTOOL

NeuralNetTool is the user interface with the NeuralNet system. It loads the model and executes all the operations requested by the user.

3.2.1 Features

NeuralNetTool was designed to support the following operations:

- **Loading of large datasets efficiently:** training a model requires datasets with millions of images (for example the ImageNet dataset which we use to train the model has 1000 categories and 10 000 images per category, so 10 000 000 images in total, around 200 GB of data). We implemented a batched generator that can load images using the CPU while the GPU is training the previous batch of images.
- **Real-time processing of the dataset images:** the dataset images must be processed on-memory before feeding them to the CNN model. We have implemented both the resizing and cropping of the image to specified model input size and the subtraction of the mean pixel of the dataset for a better classification accuracy.
- **Evaluation of the model against a given dataset:** this operation computes the accuracy of the model over a given validation dataset.
- **Training of a model:** this operation loads a full training dataset and trains the model during some specified time.
- **Fine-tuning of a model:** NeuralNet can load a previously trained model and resume the training for all or some layers with modified parameters.
- **Model state extraction:** this operation can extract all the values at the output of each layer during the inference of a given input image on the model. This data is very useful to select the best configuration for the quantization of the model.
- **Model conversion:** NeuralNet can convert a given model to other formats compatible with external analysis tools. This operation supports both HDF5 and Core ML (Apple Inc., 2017) as export formats.

3.2.2 Command Line Interface

The command line interface of NeuralNetTool was designed to be extremely flexible and allow the execution of the operations needed both during training, evaluation and analysis of a neural network model.

```
NeuralNetTool.py – Pedro José Pereira Vieito – 2017
```

```
Usage:
```

```
NeuralNetTool.py evaluate [options]  
NeuralNetTool.py train [options]  
NeuralNetTool.py progressive_quantization [options]  
NeuralNetTool.py output_extraction [options]  
NeuralNetTool.py coreml_export [options]
```

```
Options:
```

```
-f=<datasets_folder>    Datasets folder [default: DataSets]  
-d=<dataset>            Dataset name [default: ilsvrc12]
```

<code>-s=<size></code>	Image target size [default: 227]
<code>-c=<colormode></code>	Color mode [default: rgb]
<code>-m=<model></code>	Pretrained model to load
<code>-e=<epochs></code>	Epochs to train in train mode [default: 2500]
<code>-q=<resolutions></code>	Quantization resolutions k:i:o
<code>--phase=<epochs></code>	Epochs per phase in progressive_quantization
<code>--msb=<msb_percentile></code>	Quantization MSB percentile (95, 98, 99.9, 99.99)
<code>--neuralnet=<name></code>	NeuralNet name [default: QuantizedSqueezeNet]
<code>--gpu=<device></code>	GPU device to use [default: 0]
<code>-r, --relax</code>	Add a relaxation phase to the progressive quantization
<code>-v</code>	Verbosity level
<code>-h, --help</code>	Show this help

Code 1: Command line options of the NeuralNetTool.

NeuralNetTool can run the following command operations:

- **evaluate**: Evaluates the model accuracy against the validation dataset.
- **train**: Trains model with the training dataset during the specified epochs.
- **progressive_quantization**: Starts the quantization of a pretrained model following a predefined strategy. We will discuss how we quantized the model and with which techniques on the fifth chapter.
- **output_extraction**: Dumps an HDF5 file with the state of the outputs of each layer in the neural network during the evaluation of an input image batch.
- **coreml_export**: Converts and exports a trained model to Core ML format.

Furthermore, NeuralNetTool support a verbose mode (flag `--v`) which will dump the internal state of the model during the training and show any warnings or notices from the low-level libraries like TensorFlow or CUDA.

3.2.3 Dataset Loader

NeuralNetTool was designed to work with large datasets. It can load the ImageNet ILSVRC12 (ImageNet, 2012) dataset with over 10 million images and 1000 categories in less than one minute using Python iterators to preload images on memory in batches in parallel to the GPU operation.

It was implemented with a modified version of the Keras `DirectoryIterator` class, with support for various channel ordering (RGB vs. BGR, required by some pretrained models from Caffe) and the option to subtract the mean pixel of the dataset, which helps increasing the accuracy of the model during the training.

3.2.4 Execution Flow

The NeuralNetTool execution flow is divided in three steps:

- Loads the dataset specified from the Command Line using the `DirectoryIterator`.
- Loads the selected model and instantiates it with the input size detected from the dataset.
- Finally switches to the chosen command mode and starts the process.

The following simplified Python code exemplifies the execution flow of NeuralNetTool:

```
... # Initialization code and options parsing

# Dataset loading
train_generator, validation_generator, classes, shape = DataSets.load_dataset(dataset_name,
                                                                              dataset_size,
                                                                              color_mode=color_mode,
                                                                              datasets_folder=datasets_folder)

... # Checks that the Dataset is compatible and available

# Model loading
neuralnet = NeuralNet(categories=num_classes,
                      input_shape=shape,
                      subtype=train_generator.color_mode,
                      quantization_resolutions=quantization_resolutions,
                      quantization_msb= quantization_msb))

ongoing_model = "ModelData/NeuraNetTool_{}_Ongoing.h5".format(session_uuid)

# Functions that define the various operations

def train_model(epochs=1000, initial_epoch=0):
    # Executes the training with the options and saves the ongoing model after each epoch
    neuralnet.model.fit_generator(...)
    ... # Prints the accuracy and the details of the training

def evaluate_model():
    # Executes the evaluation with the command line options
    scores = neuralnet.model.evaluate_generator(...)
    ... # Prints the accuracy and the details of the evaluation

def progressive_quantization(epochs_per_phase, relaxation_phase=False):
    ... # Executes the training while progressively quantizing the layers

def layer_output_extraction():
    ... # Runs the model and extracts the output from each convolutional layer

def coreml_export():
    ... # Converts the model to Core ML with coremltools

# Main code
... # Prints details of the model and dataset

# Loads a pretrained model if any available
if neuralnet.trained_model_available() or pretrained_model:
    load_pretrained_weights()
```

```

# Executes the requested command
if args["evaluate"]:
    evaluate_model()
elif args["train"]:
    train_model(epochs=int(args["-e"]), initial_epoch=0)
elif args["progressive_quantization"]:
    evaluate_model()
    progressive_quantize_and_lock(int(args["--phase"]), relaxation_phase=args["--relax"])
elif args["output_extraction"]:
    layer_output_extraction()
elif args["coreml_export"]:
    coreml_export()

... # Prints the full session details and elapsed time

```

Code 2: Simplified code structure of NeuralNetTool.

3.2.5 Interoperability

The NeuralNet system was designed to be highly portable and platform-independent, hence, it only requires Python, Keras 2 and TensorFlow to run properly. These libraries and software are widely available on any modern computing platform.

Additionally, NeuralNetTool includes support to extract and export data in different formats for analysis and evaluation purposes:

- Supports saving the internal weights of the model in both Hierarchical Data Format (HDF5) and NumPy format. Useful for analysis in numerical computing software like MATLAB.
- Provides the option to extract the output values of each layer during inference in HDF5 for analysis and debugging.
- Includes support for exporting complete models in Core ML format to run them on Apple platforms with full hardware acceleration support.

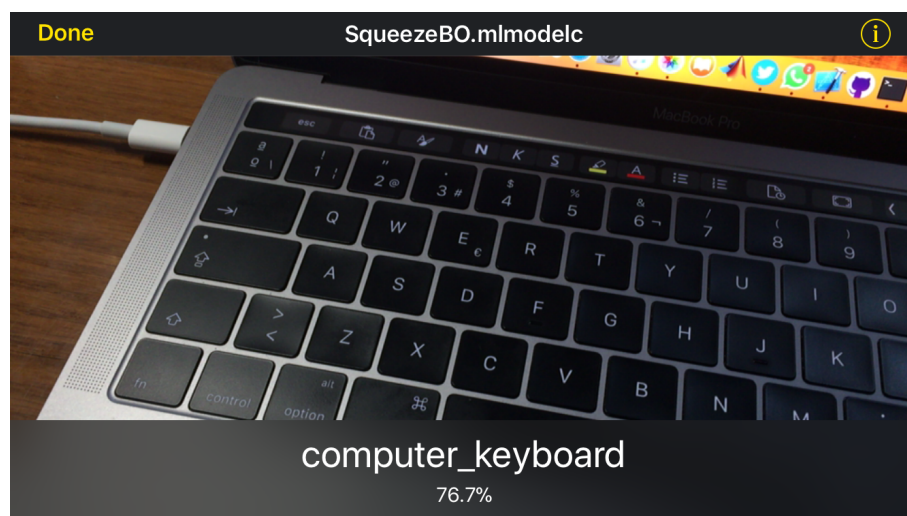


Figure 9: Model trained with NeuralNet running on DeepVision with hardware acceleration on iOS after exporting it in Core ML.

3.3 NEURALNETMODEL

Each model that follows the NeuralNetModel architecture inherits its interface from a common basic Python class: `GenericNet`. This class implements the basic code indispensable for the initialization and data storage of the model and defines the minimum requirements to interoperate successfully with NeuralNetTool.

3.3.1 *GenericNet*

`GenericNet` subclasses can override the following methods:

- `__init__(...)`: Optional customization of the model initialization.
- `create_model()`: Required override, the default code will throw an exception. All `GenericNet` subclasses are required to store their own neural network topology as a `Keras Model` instance in their `model` property.
- `compile()`: Optional customization of the model optimizer. By default, `GenericNet` implements a standard SGD optimizer.

The following code shows a simplified version of the `GenericNet` class implementation:

```
class GenericNet():

    # Common initialization method
    def __init__(self,
                  categories=1,
                  input_shape=(3, 3, 3),
                  subtype=None,
                  quantization_resolutions=None,
                  quantization_msb=None):

        # Autonoming, for example: GenericNet_227x227x3_1000_RGB
        if not self.name:
            self.name = self.__class__.__name__ +
                        "_" + str("x".join(map(str, input_shape))) +
                        "_" + str(categories)

        if subtype:
            self.name = self.name + "_" + subtype

        # Basic stored properties
        self.categories = categories
        self.input_shape = input_shape
        self.image_channels = input_shape[0]
        self.image_size = input_shape[-2:]

        # Advanced quantization properties
        self.quantization_resolutions = quantization_resolutions
        self.quantization_msb = quantization_msb

        ...
```



```

        # After all properties are initialized, GenericNet creates and compiles the model
        self.create_model()
        self.compile()

    # Required method. Should save a Keras Model in self.model
    def create_model(self):
        raise Exception("create_model() should be overridden in a GenericNet subclass")

    # Compiles the model with the default SGD Optimizer
    def compile(self):
        sgd = SGD(lr=0.04, decay=0.0002, momentum=0.9 , nesterov=True)
        self.model.compile(optimizer=sgd, loss="categorical_crossentropy",
                           metrics=["accuracy"])

    ... # Auxiliary methods to load and store the model from disk

```

Code 3: Simplified implementation of the `GenericNet` class.

3.3.2 Models

To verify the design of the NeuralNet system, we implemented some neural network models and compared the results with the ones achieved on other software. In particular, we implemented four models with the NeuralNetModel architecture:

- **BasicNet:** A sample model designed to be easily trained. This model includes two input convolution layers, two output dense layers and a pooling layer in the middle. It can quickly achieve good accuracies —up to 60%— on simple datasets like CIFAR-10 (32×32 images of 3 channels classified in 10 categories). Unfortunately, the use of 2 dense layers implies an extremely huge number of trainable parameters and a big model size —17 MB, which is huge for CIFAR-10.
- **LeNet:** The most traditional convolutional neural network. It was designed for character recognition. After training it with the MNIST dataset (16×16 images of black and white hand-written digits from the 0 to 9) we achieved an accuracy of 98.49% with a model size of 5 MB.
- **SqueezeNet:** A modern small and efficient neural network with high accuracy. This model was replicated using the Keras API following the official description. The topology and architecture of this model will be described in detail in the next chapter.
- **QuantizedSqueezeNet:** A convolutional neural network based on SqueezeNet with quantized convolutions support to adapt a full-resolution model to a low-resolution environment like the Hardware Convolution Engine of the PULP platform. The design, implementation and evolution of this model is explained in following chapters.

3.4 NEURALNETCORE

NeuralNetCore is the piece of software that must forward all the convolution operations executed by the TensorFlow framework to the Hardware Convolution Engine provided by the PULP

platform. This software is not yet implemented, but its purpose is replaced on other platforms with the hardware acceleration provided by the CUDA framework.

Finally, on platforms that do not support neither of those technologies, TensorFlow can downgrade to direct CPU computation.

4 SQUEEZE NET DESIGN

4.1 INTRODUCTION

We chose SqueezeNet as our base neural network model due to its efficient topology and small size (Iandola, Moskewicz, Ashraf, Han, & Keutzer, 2016). Its model weight data is around 5 MB and its accuracy on the ImageNet ILSVRC12 dataset is around 57.5%, extremely high for its size and number of parameters. For these properties, it is a great choice for a low-power and memory constrained platform like PULP.

Moreover, SqueezeNet was designed to require a lot of convolution operations with small size kernels (1×1 and 3×3) with non-unity strides to avoid pooling layers. This design aligns perfectly with our low-resolution HWCE. Unfortunately, SqueezeNet was designed to be trained with single-precision floating-point parameters (32-bit) thus, it will require some adjustments to adapt it to a low-resolution environment. We will describe some strategies and their results in the next chapter.

The following sections describe the original SqueezeNet design, our implementation and the modifications required to simulate a low-resolution environment.

4.2 TOPOLOGY

4.2.1 Strategies

The original designers of SqueezeNet identified three main strategies to minimize the number of parameters required by the model:

- **Replace 3×3 filters with 1×1 filters:** They chose to implement 1×1 filters whenever possible as this kernel is 9 times smaller than the 3×3 one.
- **Decrease the number of channels before a 3×3 filter:** As the number of parameters grows proportionally to channels of the convolutional layer input is critical to reduce them before any convolution with a 3×3 kernel.
- **Downsample in the last layers of the network:** As described in the second chapter, modern CNN models use convolutional layers with a stride higher than one instead of pooling layers to downsample the input data. Delaying that downsampling until the last layers has been found to lead to better accuracy on classification models (He, Zhang, Ren, & Sun, 2015).

With the first two strategies aimed at reducing the number of parameters of the model and the last one to enhance the accuracy of model the designers, created the basic piece of the SqueezeNet model, the *Fire* module.

4.2.2 Fire Module

In neural network terminology, a module is a group of basic layers that are used as a building block to create a model. The SqueezeNet model includes 10 *Fire* modules intercalated with some pooling layers. Each *Fire* module is comprised of 3 convolutional layers, each one followed by a ReLU activation layer. The input of the *Fire* module is fed into the first convolutional layer with a 1×1 filter followed by its ReLU activation layer. This initial group is called *Squeeze* layer of the *Fire* module. The output of the *Squeeze* layer is then fed in parallel to two convolutional layers, one

with a 1×1 filter and other with a 3×3 filter, each one followed by their ReLU activation layer. This group of 2 convolutional layers and 2 activations in parallel is called the *Expand* layer. Finally, both results are merged into a concatenation producing the output result of the *Fire* module.

The following schematic shows the structure of the *Fire* module, with the *Squeeze* and the *Expand* layers clearly separated.

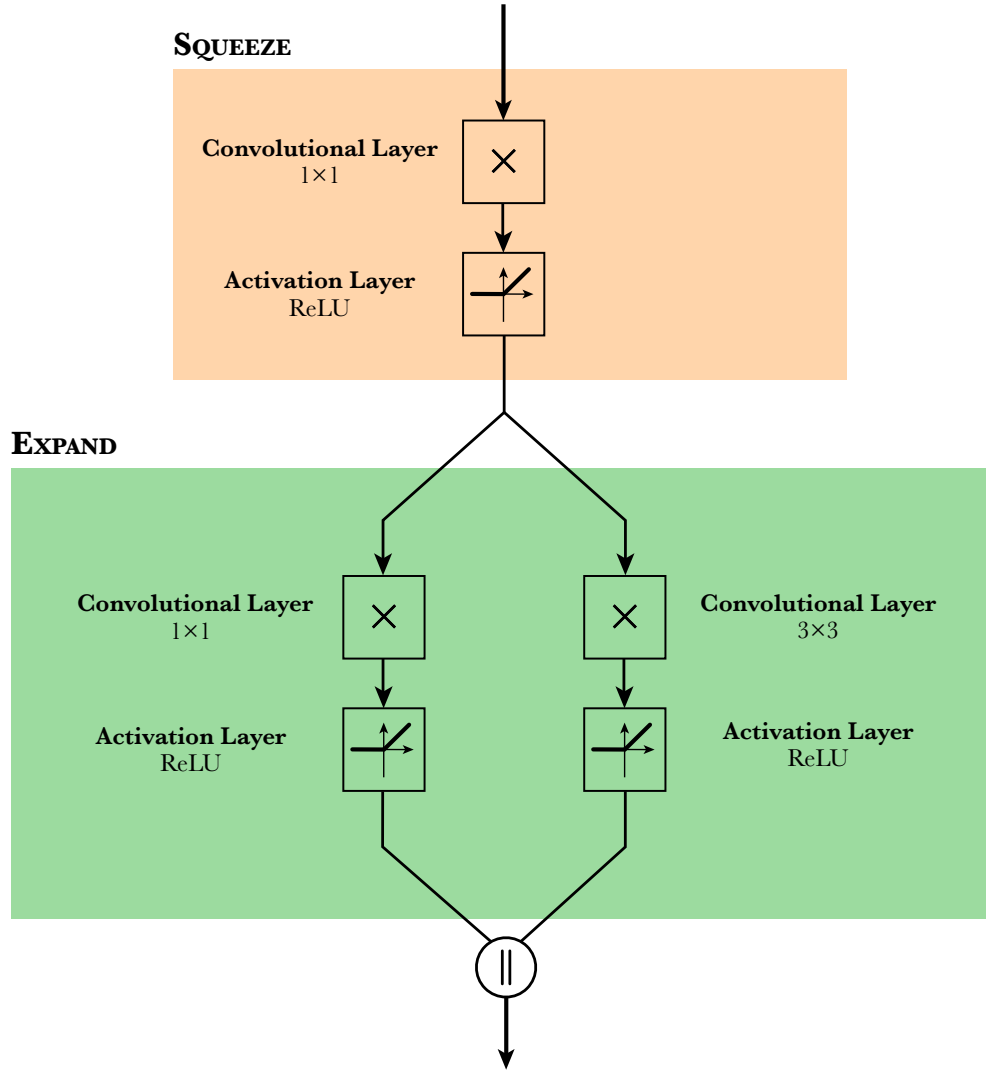


Figure 10: Internal structure of the *Fire* module.

The *Fire* module implements the first strategy of the SqueezeNet team, using two 1×1 kernels and only one 3×3 filter. Both the *Squeeze* and the *Expand* groups have some hyperparameters that can be tuned during the design of the network. In particular, each convolution can have a specified number of filters applied to its input: $s_{1 \times 1}$, $e_{1 \times 1}$ and $e_{3 \times 3}$.

Following the second strategy, the number of filters in *Squeeze* group should be less than the ones in the *Expand* one:

$$s_{1 \times 1} < e_{1 \times 1} + e_{3 \times 3}$$

This way the *Squeeze* convolution will reduce the number of output channels available as input to the 3×3 convolution, helping to reduce the number 3×3 filters which account for the vast majority of the *Fire* module trainable parameters.

4.2.3 Architecture

The SqueezeNet neural network model is designed to have 227×227 images with 3 channels as input and 1000 category probabilities as output.

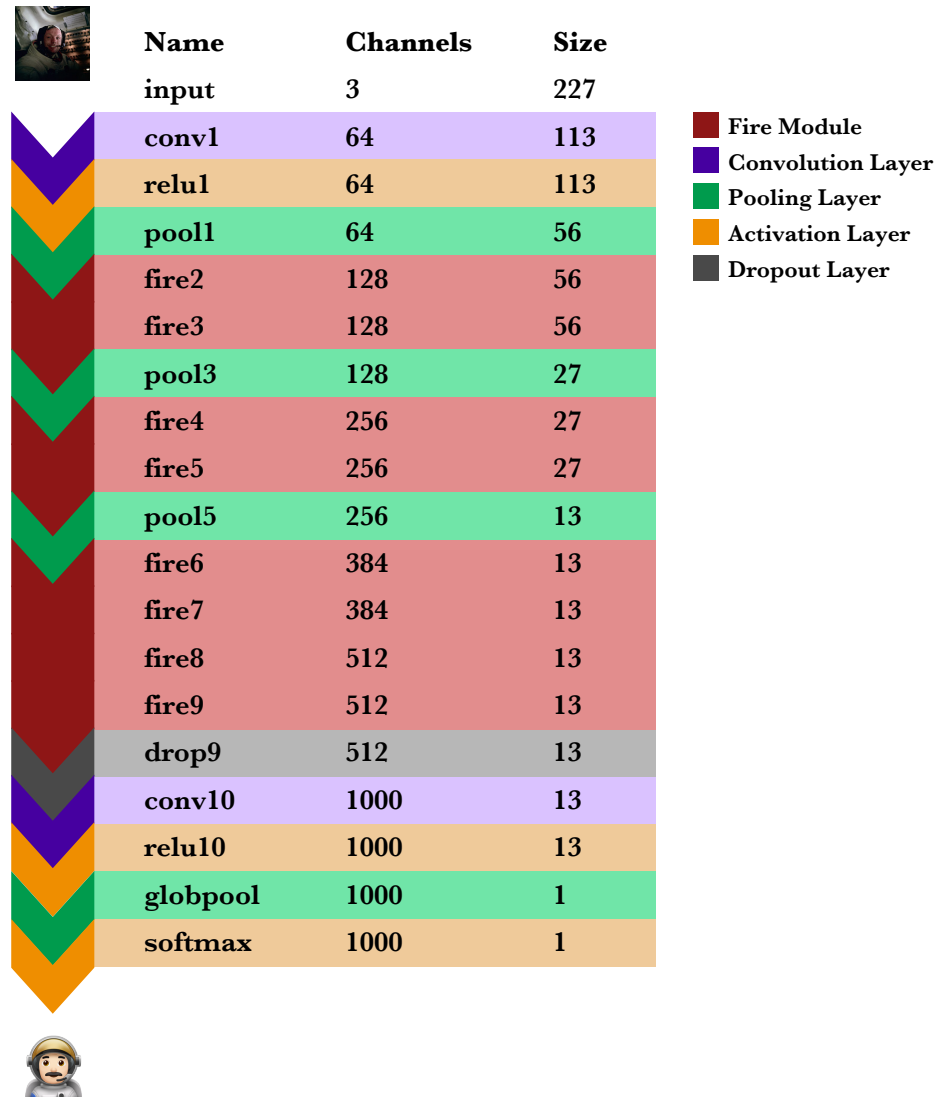


Figure 11: Structure of the full SqueezeNet neural network.

The first layer of SqueezeNet is the initial convolutional layer, followed by its ReLU activation and a pooling layer. These three layers convert the input image ($3 \times 227 \times 227$) on 64 images of 56×56 pixels. The process of downsampling and converting the spatial information on feature

resolution (channels) in SqueezeNet is very gradual and performed by the first *Fire* modules of each pair (`fire2`, `fire4`, `fire6`, `fire8`).

Following all the Fire modules and before the last convolutional layer there is a `dropout` layer. This layer is only active during training and it is designed to nullify a specified percentage of its input values. SqueezeNet defines that 50 percent of the values should be dropped. This type of layer is implemented to avoid overfitting the model during training — feeding so many data that the model only learns to classify the training set, not general images.

The last feature extractor of the network is a convolutional layer (`conv10`) which outputs the 1000 categories fed through a Global Average Pooling (`globpool`), which averages each one of the 1000 13×13 matrices to its mean value.

The final layer is a `softmax`, a special activation function which converts any k -dimensional vector of real values to a k -dimensional vector with values in the range $[0, 1]$ that add up to 1. This means that the output values can be treated directly as normalized probabilities which is a great feature for an image classifier.

4.3 IMPLEMENTATION

To recreate the SqueezeNet model we followed the Caffe model description available in the official GitHub repository (DeepScale, 2016). We selected the version v1.1 of SqueezeNet which uses 41% of the computation required by the initial version while obtaining the same accuracy.

The published Caffe models include two files:

- A `caffemodel` file which stores the weights after the training in a binary form. We converted this data to an HDF5 file which is loadable by Keras.
- A `prototxt` file which declares the structure and layers of the model. Using this file and the description from the SqueezeNet paper we recreated the neural network topology on Keras.

The model was implemented as a `GenericNet` subclass to follow to `NeuralNetModel` architecture. As SqueezeNet does not require any special initialization or optimizer the `SqueezeNet` class does only override the `create_model()` function from `GenericNet`.

4.3.1 Model Tensor

To understand the implementation of the model, it is important to highlight how Keras and TensorFlow compute the values of the model internally. When a new model is build, Keras instantiates a TensorFlow graph that represents the connection between its input and output. This graph, called in TensorFlow terms *tensor*, includes all the operations done in between the nodes. This approach allows the software to use the tensor both to compute the output for a given input or the gradient while back-propagating the error during the training.

4.3.2 Fire Module

The *Fire* module is the basic building block of the SqueezeNet model. We have implemented this module as a Python function that appends the *Fire* module operations to an input tensor:

```
def fire_module(input, squeeze, expand, name="fire"):
    x = Conv2D(squeeze, (1, 1), padding="valid",
```

```

        activation="relu",
        name=name + "/squeeze1x1")(input)

    branch_1 = Conv2D(expand, (1, 1), padding="valid",
        activation="relu",
        name=name + "/expand1x1")(x)
    branch_2 = Conv2D(expand, (3, 3), padding="same",
        activation="relu",
        name=name + "/expand3x3")(x)

    return concatenate([branch_1, branch_2], axis=3, name=name + "concat")

```

Code 4: Function `fire_module` that appends the *Fire* module operators to an input tensor.

The `fire_module` function includes the `squeeze` and `expand` parameters to define the number of *Squeeze* and *Expand* filters applied on each module of the SqueezeNet neural network.

4.3.3 SqueezeNet

Using the `fire_module` function and the built-in Keras layers, we defined the `SqueezeNet` class with the following Python code:

```

class SqueezeNet(GenericNet):

    def create_model(self):

        def fire_module(input, squeeze, expand, name="fire"):
            ... # Code 8 snippet.

        input = Input(shape=self.input_shape)

        x = Conv2D(64, (3, 3), strides=(2, 2), activation="relu", name="conv1")(input)

        x = MaxPooling2D(pool_size=(3, 3), strides=(2, 2), name="pool1")(x)

        x = fire_module(x, squeeze=16, expand=64, name="fire2")
        x = fire_module(x, squeeze=16, expand=64, name="fire3")

        x = MaxPooling2D(pool_size=(3, 3), strides=(2, 2), name="pool3")(x)

        x = fire_module(x, squeeze=32, expand=128, name="fire4")
        x = fire_module(x, squeeze=32, expand=128, name="fire5")

        x = MaxPooling2D(pool_size=(3, 3), strides=(2, 2), name="pool5")(x)

        x = fire_module(x, squeeze=48, expand=192, name="fire6")
        x = fire_module(x, squeeze=48, expand=192, name="fire7")
        x = fire_module(x, squeeze=64, expand=256, name="fire8")
        x = fire_module(x, squeeze=64, expand=256, name="fire9")

        x = Dropout(0.5, name="drop9")(x)

```

```

x = Conv2D(self.categories, (1, 1), activation="relu", name="conv10")(x)
x = GlobalAveragePooling2D()(x)
out = Activation("softmax", name="loss")(x)

self.model = Model(inputs=input, outputs=out)

```

Code 5: Python code that defines the SqueezeNet class.

The model definition is done on the `create_model` function from its superclass, `GenericNet`, making the model compatible with the `NeuralNetModel` architecture.

4.3.4 Verification

Once we had designed the implementation, we verified that it worked properly with the pretrained weight data. After loading the official pretrained weight model, we got a 56.9% accuracy on the ILSVRC12 verification dataset. The SqueezeNet authors state that their pretrained model has a 57.8% accuracy, so the small difference could be sourced in the unique implementation of the various algorithms used by Keras and Caffe internally. As these results were very similar, we established that the model had been correctly ported to Keras.

There are some important notes to highlight about model conversion from Caffe to Keras:

- Caffe loads the dataset images in BGR channel order instead of the universal RGB. This implies that the dataset loader of the Keras implementation must be able to swap the channels of the images while loading them from disk.
- The SqueezeNet model published by the authors was trained on a dataset with the ImageNet ILSVRC12 mean pixel subtracted. This requires the dataset loader to apply the same transformation when loading the verification images.

Avoiding the mean pixel subtraction reduces the verified accuracy of the model to 36.5% and ignoring the channel swapping to 16.4%. The transformations applied to the training dataset are extremely important to be consistently repeated on other external inputs like the verification dataset, as the model has been biased to classify images which include these transformations.

4.4 QUANTIZATION

Because the objective of this work is to obtain a model designed for low-resolution convolutions, we had to implement some functions that allowed the simulation of that environment. These functions will have to quantize the inputs, execute the convolution, quantize the output and forward the result to the next layer, simulating a Hardware Convolution Engine.

Training the model with these low-resolution constrains implies that all the weight parameters applied to convolutions can be stored on-disk with the reduced resolution without varying the resulting accuracy. For example, after quantizing and training the model with 16-bit resolution values instead of 32-bit float we could save up to 50% of store space without any loss on the accuracy. This property of quantized models must be balanced with the fact that low-resolution convolutions on neural networks require more complex and time-consuming training methods.

4.4.1 Simulated Quantization

The implementation of those low-resolution convolutions on the model was accomplished by creating a new `NeuralNetModel` called `QuantizedSqueezeNet` and a subclass of the built-in Keras `Conv2D` layer with quantization support, `QuantizedConv2D`. The main code of the `QuantizedSqueezeNet` class is similar to `SqueezeNet` but replacing each instance of `Conv2D` by a `QuantizedConv2D`.

`QuantizedConv2D` was designed to be completely configurable. Both the bit resolution and the most significant bit of the input, kernel and output can be independently tuned. The layer also includes the option to bypass the quantizations, behaving exactly like a `Conv2D` layer.

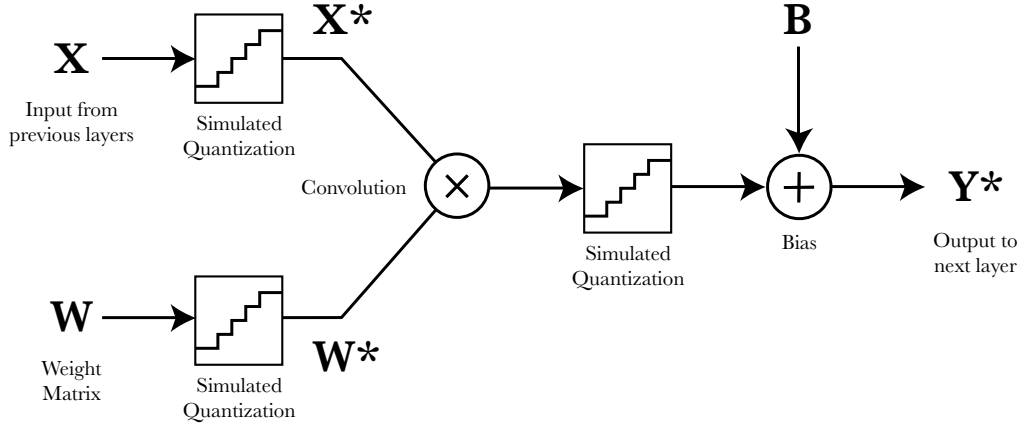


Figure 12: Structure of the `QuantizedConv2D` layer.

The core of this new layer is the function block that simulates a quantization of the inputs and output of the convolution. This operation must clip the values around a specified range and quantize the result. The clipping range is given by the most significant bit (MSB) of the quantization configuration:

$$A = \pm 2^{MSB}$$

Equation 5: Formula to compute the amplitude of the clipping range from the specified MSB.

The resolution of quantization can be calculated from this result and the number of bits, n :

$$q = \frac{2 * |A|}{2^n} = \frac{2^{MSB+1}}{2^n}$$

Equation 6: Relation between the quantization resolution, MSB and number of bits of the parameter.

The complete quantization operation would proceed the following way. First, it computes amplitude of the clipping boundaries:

$$A = 2^{MSB}$$

Equation 7: Obtaining the clipping amplitude from the MSB parameter.

Then, the quantization resolution:

$$q = \frac{2 * A}{2^n}$$

Equation 8: Computing the quantization resolution from the amplitude and the number of bits.

And with the amplitude, it can get the clipped value, x_c , of the input, x :

$$x_c = \min(\max(x, -A), A)$$

Equation 9: Clipping the input to the amplitude boundaries.

Finally, it will compute the output by dividing the clipped value between the quantization resolution, flooring the result and multiplying again by the resolution to recover the quantized value, x_q :

$$x_q = \left\lfloor \frac{x_c}{q} \right\rfloor * q$$

Equation 10: Quantizing the clipped value with the computed resolution.

4.5 QUANTIZATION IMPLEMENTATION

4.5.1 Quantclip Operator

To implement these equations, we defined a function called `quantclip` that receives an input tensor, the MSB and resolution parameters and returns an extended tensor with the quantization operations appended to it. To add these operations to the input tensor, we had to implement it using the basic built-in operators from TensorFlow and Keras the following way, being x the input tensor, `msb` the most significant bit binary value (0 would mean that the MSB value is 2^0) and `nbits` the resolution of the quantization:

```
from keras import backend as K
import tensorflow as tf

def quantclip(x, msb, nbits):
    # Amplitude
    amplitude = 2 ** msb

    # Clipping
    x = K.clip(x, -amplitude, amplitude)

    # Resolution (for 2's complement)
    q = tf.constant(float(2 * amplitude) / float(2 ** nbits))

    # Quantization
    y = tf.scalar_mul(q, tf.floordiv(x, q))
    return y
```

Code 6: Python code that defines a function that appends the quantization operations to the output of a tensor using Keras and TensorFlow operators.

The previous code works properly when a tensor is fed with an input value, but will fail while computing its gradient. This is sourced in the definition of the `floordiv` operator in TensorFlow, which returns `None` as its constant gradient. While this behavior can be opportune in some cases, as the real gradient of the floor operation is usually 0 or not defined, during back-propagation this result breaks the gradient computation. It is interesting to note that this incompatibility is somewhat similar to one found in Binarized Neural Networks, which apply a binarization function to the values of the model (Courbariaux, Hubara, Soudry, El-Yaniv, & Bengio, 2016). To solve this restriction, we applied a solution similar to the referenced case, overriding the floor function gradient with the Identity case:

```
from keras import backend as K
import tensorflow as tf

def quantclip(x, msb, nbits):

    # Amplitude
    amplitude = 2 ** msb

    # Clipping
    x = K.clip(x, -amplitude, amplitude)

    # Resolution (for 2's complement)
    q = tf.constant(float(2 * amplitude) / float(2 ** nbits))

    # Quantization
    d = tf.div(x, q)

    # Overriding the gradient value of the Floor operator
    with tf.get_default_graph().gradient_override_map({"Floor": "Identity"}):
        f = tf.floor(d)

    # Computing the output
    y = tf.scalar_mul(q, tf.floordiv(x, q))
    return y
```

Code 7: Extended Python code that defines a function that appends the quantization operators to an input tensor and supports the computation of gradients.

4.5.2 *QuantizedConv2D*

The `quantclip` function is the main building block to convert a basic `Conv2D` layer into `QuantizedConv2D`. We achieved this by subclassing Keras `Conv2D` class and modifying its `__init__()` and `call()` functions.

The initialization function was extended to include all the configuration parameters requested by the customizable `quantclip` operator (`quantization_resolutions`, `quantization_msb`), an option to bypass the quantization (`quantization_enabled`) and a parameter to extract the convolution raw data before the activation and bias (`perform_extraction`).

```
def __init__(self, filters, kernel_size,
```

```

        quantization_resolutions,
        quantization_msb,
        quantization_enabled=True,
        perform_extraction=False,
        **kwargs):
    self.quantization_enabled = quantization_enabled
    self.quantization_resolutions = quantization_resolutions
    self.quantization_msb = quantization_msb
    self.perform_extraction = perform_extraction

    super(QuantizedConv2D, self).__init__(filters, kernel_size, **kwargs)
...

```

Code 8: QuantizedConv2D initialization code with the extended parameters.

The `quantization_resolutions` and `quantization_msb` parameters are Python lists with 3 values that store the resolution and MSB of the convolution kernel, input and output. For example, the following code initializes a `QuantizedConv2D` layer with a 16-bit kernel, 8-bit input and 6-bit output with all MSB representing 2^0 .

```

num_filters = 64
kernel_size = (3, 3)
quant_res = [16, 8, 6]
quant_msb = [0, 0, 0]

myQuantizedLayer = QuantizedConv2D(num_filters, kernel_size, quant_res, quant_msb)

```

Code 9: Python code that instantiates a `QuantizedConv2D` layer.

The second modification required in `QuantizedConv2D` is to add the `quantclip` operator to the `call()` function, which is ran while Keras creates the tensor of the model. The default implementation of `Conv2D` uses its internal superclass `_Conv` to execute the following code that appends a 2-dimensional convolution, a bias and an activation function on the input:

```

def call(self, inputs):
    outputs = K.conv2d(
        inputs,
        self.kernel,
        strides=self.strides,
        padding=self.padding,
        data_format=self.data_format,
        dilation_rate=self.dilation_rate)

    if self.use_bias:
        outputs = K.bias_add(
            outputs,
            self.bias,
            data_format=self.data_format)

    if self.activation is not None:

```

```

        return self.activation(outputs)
    return outputs

```

Code 10: Simplified code from the internal `_Code` layer class of Keras (Chollet, 2015).

Our implementation redefines this function, maintaining the basic structure and adding the quantization of the input, kernel and output around the `k.conv2d()` function with the output extraction support:

```

class QuantizedConv2D(Conv2D):

    def __init__(...):
        ... # Code 8 snippet.

    def quantclip(self, x, msb, nbits):
        ... # Code 7 snippet.

    def call(self, inputs):

        kernel = self.kernel

        if self.quantization_enabled:
            kernel_bits = self.quantization_resolutions[0]
            input_bits = self.quantization_resolutions[1]
            output_bits = self.quantization_resolutions[2]
            kernel_msb = self.quantization_msb[0]
            input_msb = self.quantization_msb[1]
            output_msb = self.quantization_msb[2]

            # Quantization of the input and kernel
            inputs = self.quantclip(inputs, input_msb, input_bits)
            kernel = self.quantclip(kernel, kernel_msb, kernel_bits)

        outputs = K.conv2d(
            inputs,
            kernel,
            strides=self.strides,
            padding=self.padding,
            data_format=self.data_format,
            dilation_rate=self.dilation_rate)

        if self.quantization_enabled:
            # Quantization of the output
            outputs = self.quantclip(outputs, output_msb, output_bits)

        if not self.perform_extraction:
            if self.use_bias:
                outputs = K.bias_add(
                    outputs,

```

```

        self.bias,
        data_format=self.data_format)

    if self.activation is not None:
        return self.activation(outputs)

    return outputs

```

Code 11: Code that defines the QuantizedConv2D class with the main changes with Conv2D highlighted.

4.5.3 Quantized Fire Module

We updated the `fire_module` function to implement QuantizedConv2D instead of Conv2D and to support the output extraction:

```

def fire_module(input, squeeze, expand,
                quantization_resolutions,
                quantization_msb,
                quantization_enabled,
                perform_extraction=False,
                name="fire"):

    x = QuantizedConv2D(squeeze, (1, 1),
                        padding="valid",
                        activation="relu",
                        name=name + "squeeze1x1",
                        quantization_enabled=quantization_enabled,
                        quantization_resolutions=quantization_resolutions,
                        quantization_msb=quantization_msb,
                        perform_extraction=perform_extraction)(input)

    # In Fire layers the output extraction is performed after the first
    # Squeeze convolution which constrains the result of the full layer
    if perform_extraction:
        return x

    else:
        branch_1 = QuantizedConv2D(expand, (1, 1),
                                    padding="valid",
                                    activation="relu",
                                    name=name + "expand1x1",
                                    quantization_enabled=quantization_enabled,
                                    quantization_msb=quantization_msb,
                                    quantization_resolutions=quantization_resolutions)(x)

        branch_2 = QuantizedConv2D(expand, (3, 3),
                                    padding="same",
                                    activation="relu",
                                    name=name + "expand3x3",
                                    quantization_enabled=quantization_enabled,

```

```

        quantization_msb=quantization_msb,
        quantization_resolutions=quantization_resolutions)(x)

    return concatenate([branch_1, branch_2], axis=3, name=name + "concat")

```

Code 12: Fire module function adapted with quantization and extraction support.

4.5.4 QuantizedSqueezeNet

The QuantizedSqueezeNet class follows the NeuralNetModel architecture in a similar way as SqueezeNet does, but replacing all the instances of Conv2D layers with QuantizedConv2D and adding the option to extract the model tensor output after any convolution.

The following Python code shows the main structure of the QuantizedSqueezeNet class with the extended create_model() method and two additional functions:

- The quantize(level) function quantizes all QuantizedConv2D layers up to the specified level, being 1 only the conv1 layer quantized, 10 all layers quantized and 0 no layers quantized.
- The trunk_at_layer(extraction_level, quantization_level) method trunks the model tensor at the specified QuantizedConv2D layer level with the quantization defined. This method is required for the output extraction mechanism of NeuralNetTool, useful to find the best quantization parameters.

```

class QuantizedSqueezeNet(GenericNet):

    # Quantization Support
    def quantize(self, level):
        self.create_model(quantization_level=level)
        self.compile()

    # Output Extraction Support
    def trunk_at_layer(self, extraction_level, quantization_level=10):
        self.create_model(quantization_level=quantization_level,
                           extraction_level=extraction_level)
        self.compile()

    def create_model(self, quantization_level=10, extraction_level=11):

        def fire_module(...):
            ... # Code 12 snippet.

        if self.quantization_resolutions is None or self.quantization_msb is None:
            quantization_level = 0 # Do not quantize without quantization parameters

        input = Input(shape=self.input_shape)
        x = input # Required for input extraction (extraction_level == 0)

```

```

if extraction_level >= 1:
    x = QuantizedConv2D(64, (3, 3), strides=(2, 2), padding="valid", name="conv1",
        activation="relu",
        quantization_enabled=(quantization_level >= 1),
        quantization_resolutions=self.quantization_resolutions,
        quantization_msb=self.quantization_msb[0],
        perform_extraction=(extraction_level == 1))(input)

if extraction_level >= 2:
    x = MaxPooling2D(pool_size=(3, 3), strides=(2, 2), name="pool1")(x)
    x = fire_module(x, fire_id=2, squeeze=16, expand=64,
        quantization_enabled=(quantization_level >= 2),
        quantization_resolutions=self.quantization_resolutions,
        quantization_msb=self.quantization_msb[1],
        perform_extraction=(extraction_level == 2))

if extraction_level >= 3:
    x = fire_module(x, fire_id=3, squeeze=16, expand=64,
        quantization_enabled=(quantization_level >= 3),
        quantization_resolutions=self.quantization_resolutions,
        quantization_msb=self.quantization_msb[2],
        perform_extraction=(extraction_level == 3))

if extraction_level >= 4:
    x = MaxPooling2D(pool_size=(3, 3), strides=(2, 2), name="pool3")(x)
    x = fire_module(x, fire_id=4, squeeze=32, expand=128,
        quantization_enabled=(quantization_level >= 4),
        quantization_resolutions=self.quantization_resolutions,
        quantization_msb=self.quantization_msb[3],
        perform_extraction=(extraction_level == 4))

if extraction_level >= 5:
    x = fire_module(x, fire_id=5, squeeze=32, expand=128,
        quantization_enabled=(quantization_level >= 5),
        quantization_resolutions=self.quantization_resolutions,
        quantization_msb=self.quantization_msb[4],
        perform_extraction=(extraction_level == 5))

if extraction_level >= 6:
    x = MaxPooling2D(pool_size=(3, 3), strides=(2, 2), name="pool5")(x)
    x = fire_module(x, fire_id=6, squeeze=48, expand=192,
        quantization_enabled=(quantization_level >= 6),
        quantization_resolutions=self.quantization_resolutions,
        quantization_msb=self.quantization_msb[5],
        perform_extraction=(extraction_level == 6))

if extraction_level >= 7:
    x = fire_module(x, fire_id=7, squeeze=48, expand=192,
        quantization_enabled=(quantization_level >= 7),

```



```

        quantization_resolutions=self.quantization_resolutions,
        quantization_msb=self.quantization_msb[6],
        perform_extraction=(extraction_level == 7))

    if extraction_level >= 8:
        x = fire_module(x, fire_id=8, squeeze=64, expand=256,
            quantization_enabled=(quantization_level >= 8),
            quantization_resolutions=self.quantization_resolutions,
            quantization_msb=self.quantization_msb[7],
            perform_extraction=(extraction_level == 8))

    if extraction_level >= 9:
        x = fire_module(x, fire_id=9, squeeze=64, expand=256,
            quantization_enabled=(quantization_level >= 9),
            quantization_resolutions=self.quantization_resolutions,
            quantization_msb=self.quantization_msb[8],
            perform_extraction=(extraction_level == 9))

    if extraction_level >= 10:
        x = Dropout(0.5, name="drop9")(x)
        x = QuantizedConv2D(self.categories, (1, 1), padding="valid", name="conv10",
            activation="relu",
            quantization_enabled=(quantization_level >= 10),
            quantization_resolutions=self.quantization_resolutions,
            quantization_msb=self.quantization_msb[9],
            perform_extraction=(extraction_level == 10))(x)

    if extraction_level >= 11:
        x = GlobalAveragePooling2D()(x)
        x = Activation("softmax", name="loss")(x)

    self.model = Model(inputs=input, outputs=x)

```

Code 13: QuantizedSqueezeNet definition with the main differences with SqueezeNet highlighted.

The quantization is implemented by passing the quantization parameters to the QuantizedConv2D layers directly or through the `fire_module` functions. The extraction mechanism is designed to stop extending the network at the specified level; thus, returning the trunked model to NeuralNetTool which would extract the output at that point while feeding some images to the input.

5 EXPERIMENTAL RESULTS

This chapter describes the steps followed to adapt the model to a low-resolution environment while highlighting the results obtained along the process. The main phases were following:

- Train the original model converted to Keras from scratch using a complete ILSVRC-2012 dataset and obtain a similar accuracy to the one achieved by the SqueezeNet authors.
- Analyze the trained model data to define the best configuration for the quantization parameters.
- Quantize the model data externally with various configurations and evaluate them on the system.
- Try various parameter configurations with the on-model quantization and compare the results with the off-model quantization.
- Select the best configuration and try some dynamic quantization techniques to enhance the accuracy of model on a low-resolution environment.

5.1 TRAINING

5.1.1 Training Setup

To ensure we have designed a good environment for training the model we tried to reproduce the original results using the same full dataset. We configured the various parameters of the training following the specifications published by the SqueezeNet team as a Caffe solver (`solver.prototxt`):

- Learning Rate: 0.04
- Batch size: 512 images
- Training size: 170 000 batches — 87 040 000 image iterations

Usually, the length of a training is defined in epochs, we have chosen to define an epoch as 1000 image iterations. Each epoch training the SqueezeNet model on a server provided with an Nvidia Titan X GPU, takes around 200 seconds to process. Unfortunately, we would require months of processing time to achieve the same level of training stated in the Caffe solver; therefore, we have set our goal at 50% accuracy and around 5000 epochs of training — 12 days of training time.

The solver configuration specifies a linearly-decaying learning rate. As we could not train the model during all the epochs specified, we decided to use a SGD optimizer with exponential learning rate decay. We configured its parameters as follows:

- Base Learning Rate: 0.04 — The one defined on the original Caffe solver.
- Decay: 0.0002 — A low decay to simulate a linearly decaying rate with small slope.
- Momentum: 0.9 — The default momentum of the SGD optimizer.

These configuration parameters were implemented as a custom optimizer in the SqueezeNet class following the NeuralNetModel architecture, as explained in section 3.3.1.

5.1.2 Training Results

After 5000 epochs of training, the SqueezeNet model achieved an accuracy of around 51%. This result allowed us to start analyzing the trained model data while providing a custom model to adapt to a low-resolution environment. Achieving an accuracy similar to the original model, 56,7%, would probably require more training time and a more similar implementation of the Caffe linearly-decaying optimizer.

The following graph shows the evolution of the accuracy of the model during the full training.

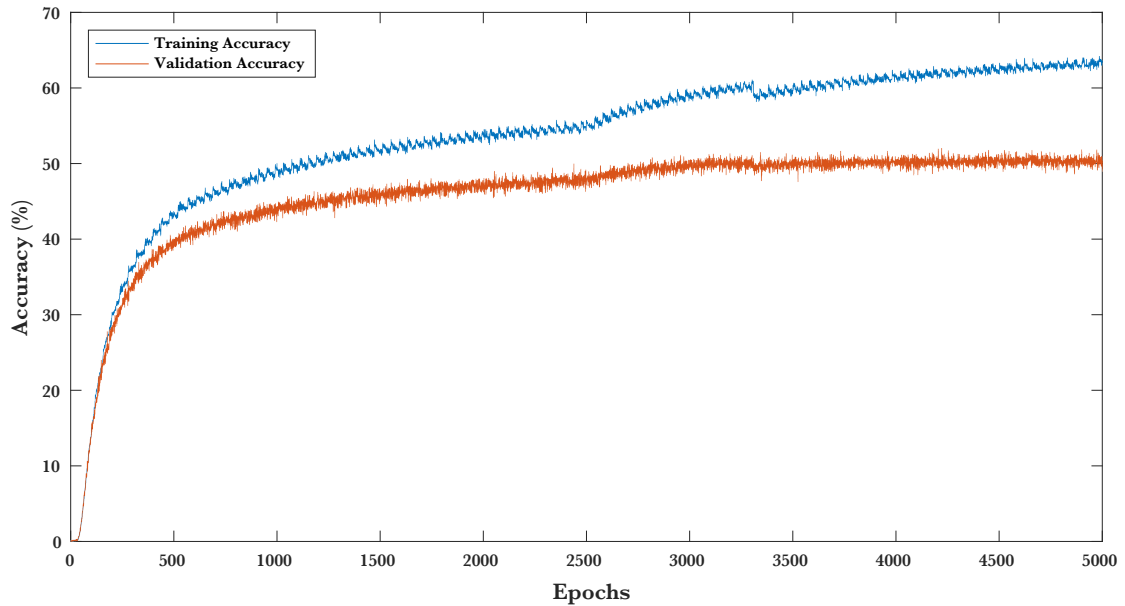


Figure 13: Evolution of the accuracy of the model during the training.

The difference between the training and validation accuracy is that the first one is computed against the training dataset while the second one uses an external validation set. The validation accuracy is the most valuable statistic as it indicates the ability of the model to predict the class of any general image, not only the ones from the training set.

The full training was segmented in three different stages:

- From the epoch 0 to 2500 it used the original setup.
- On the epoch 2500 we tweaked the learning rate as the model accuracy was stagnating. The SGD optimizer was reset with a base learning rate of 0.0037.
- At epoch 3300 we reset the optimizer with a base learning rate of 0.0033 to avoid overfitting as only the training accuracy was increasing.

5.2 MODEL DATA ANALYSIS

We should recall that the Hardware Convolution Engine of the PULP platform requires low-resolution inputs (4, 6, 8 or 16 bits) instead of the typical 32-bit floating-point values. To adapt our model to this constrained environment while maintaining a high accuracy we had to choose a correct set of quantization parameters. To do so, we analyzed the model data obtained during the full training and with the output extraction mechanism of NeuralNetTool.

Every convolutional layer on the quantized model has three quantization operations, each one with two parameters: resolution and most significant bit.

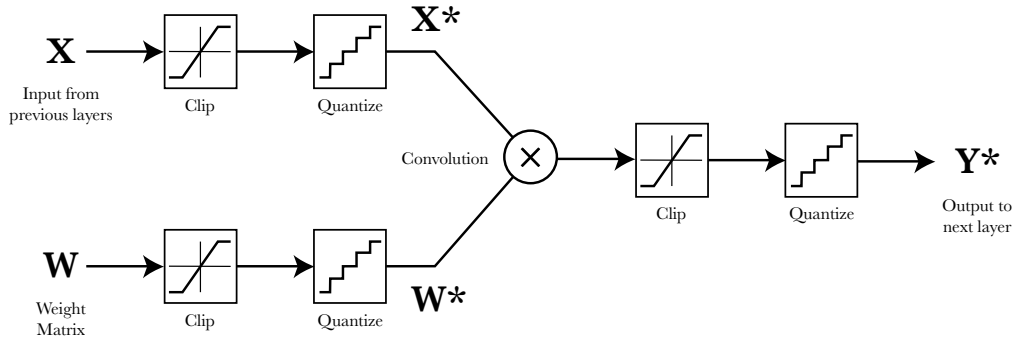


Figure 14: Structure of a convolutional layer with quantization (ignoring the biases).

5.2.1 Weigh Data Analysis

The model data file obtained during the full training contains all the kernel values from the weight matrices of each convolutional layer. We used this data to convey the best quantization parameters for the kernel input of the convolutions.

To select the best MSB value of the kernel quantization we require to know in between which boundaries are most of the kernel weight parameters. The following graph is a histogram of the kernel values of all convolutional layers in the trained model.

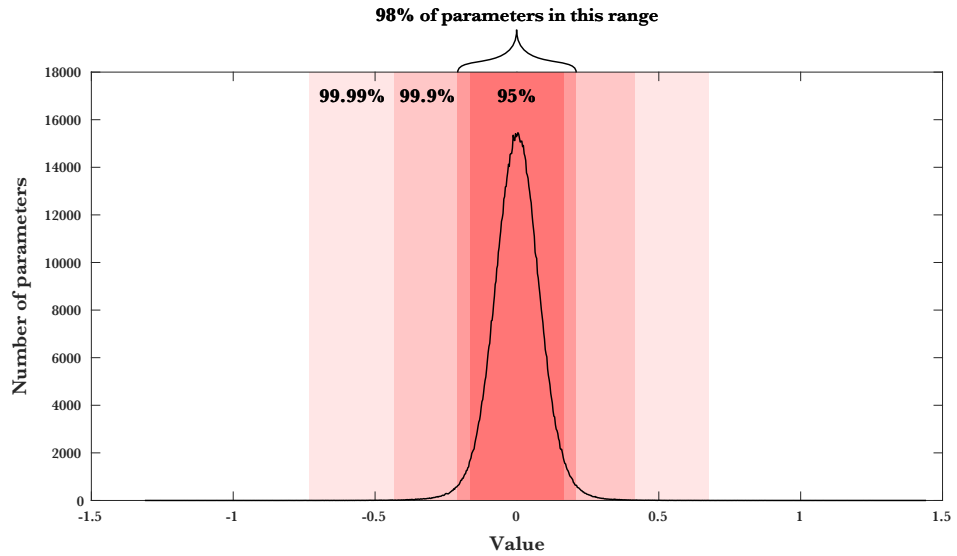


Figure 15: Histogram of all kernel parameters of the model with the four configured percentiles.

From the histogram analysis of the kernel parameters we extracted the following boundaries for four percentiles:

Percentile	Limits	
99.99 %	-0.7315	0.6775
99.9 %	-0.4321	0.4153
98 %	-0.2158	0.2088
95 %	-0.1693	0.1659

Table 2: Different value limits to fit each percentile of kernel parameters.

These limits are useful to compute a general most significant bit parameter for the kernel quantization in all layers. To better tailor this parameter for each layer, we obtained the limits for each percentile, not only for the full kernel parameters but also for each convolutional layer. The following plot shows for each percentile, the absolute maximum boundary that fits the specified percentage of the kernel parameters in each layer.

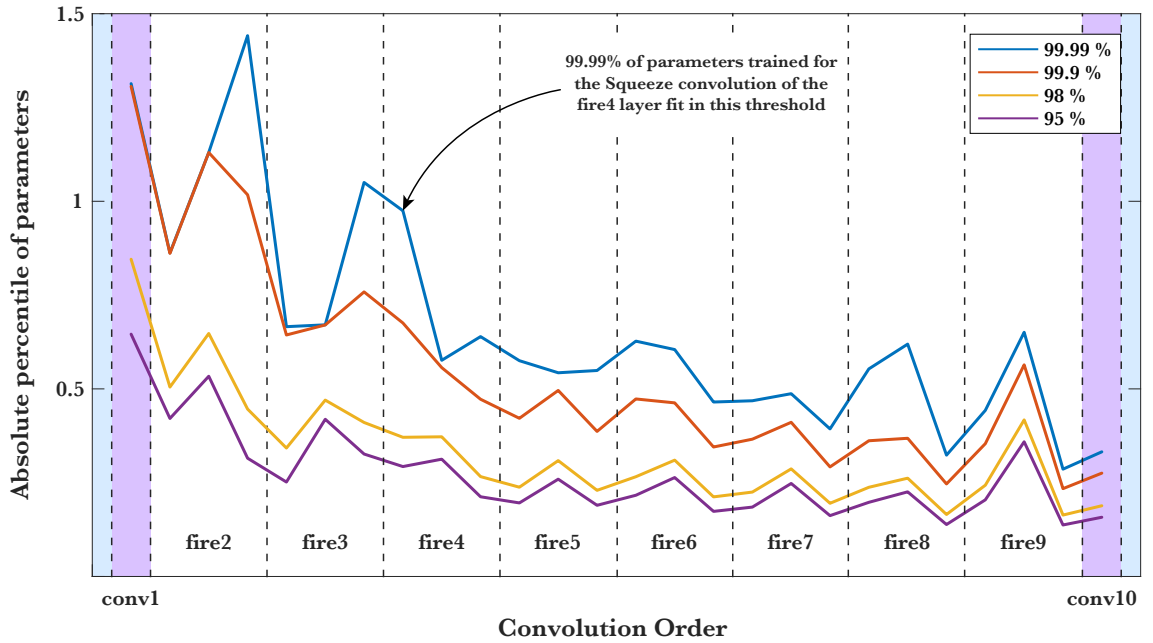


Figure 16: Limit established by each percentile on the weight data of each convolution of the model.

For example, in the conv2 layer 99.99% of the kernel parameters can be fitted in between -1.4 and 1.4 boundaries. These more detailed results allow us to compute the MSB required by each layer to fit a percentage of kernel parameters.

From these results, we can convey that the kernel parameters of most layers can fit in the ± 1 boundaries, but the first ones would require an extended margin for the most stringent percentiles.

5.2.2 Convolution Output Analysis

A similar analysis was done for the output of the convolutions. As the model data does not include any information about the value of convolution output we implemented the output extraction mechanism described in section 3.2.1. This procedure extracts the values of the model tensor at each convolution output while fed with a batch of images. To have a good statistical measure of the convolution output values for any given input, we used a batch of 32 random images from the verification dataset which produced more than 42 million output values.

The following plot shows a histogram of all the convolution output values extracted from the model during the processing of the image batch.

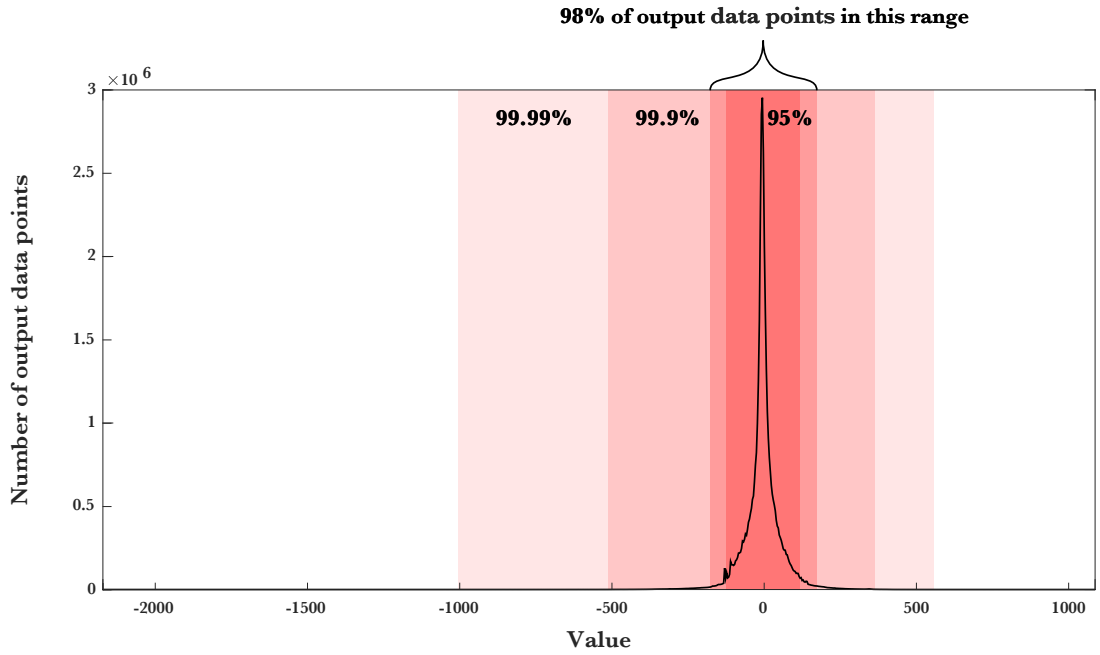


Figure 17: Histogram of all convolution output values with the four configured percentiles.

We extracted the limits for each percentile from the histogram which could be used to compute a general MSB for the quantization of all convolution outputs:

Percentile	Limits	
99.99 %	-1008	571.9
99.9 %	-507	377.8
98 %	-170	184.5
95 %	-115.8	127.3

Table 3: Different value limits to fit each percentile of convolution output data points.

As in the kernel case, we did a more exhaustive analysis of the output data separating the boundary results per layer. Note that the *Fire* modules output extraction was executed only in the first convolution (*Squeeze phase*) as referenced in Code 13.

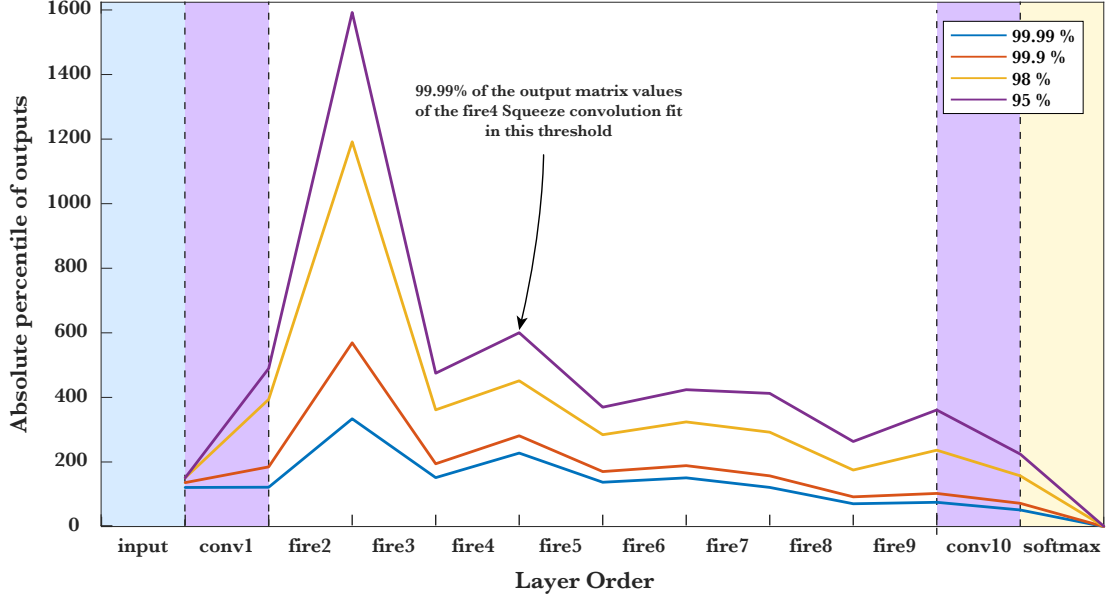


Figure 18: Maximum threshold on the output data of each convolutional layer of the model on each percentile. On the *Fire* modules only the output of the Squeeze convolution is plotted.

From these results, we can infer some satisfactory MSB configurations for each layer. It would seem that the parameters will be similar with the notable exception of the *fire2* layer which will require a bigger value. We must take this abnormality into account during the configuration of the input and output of the quantized convolutions of this layer.

It is important to highlight that this data is representative for both the output and input stages of the model convolutions. This is due to the fact that any given output is fed to the next convolutional layer only being modified by ReLU activations (which will pass through any positive values), pooling layers (which will only average the values) or biases (which are negligible in this case, typically around ± 0.5).

5.3 OFF-MODEL QUANTIZATION

This section explains how we quantized the kernel parameters and evaluated its effect for various configurations before switching to the on-model quantization implemented in *QuantizedSqueezeNet*. To do so, we loaded the trained HDF5 model data file on MATLAB, quantized the kernel parameters and saved them as a new model file which we loaded on NeuralNet to evaluate its accuracy.

5.3.1 *Quantclip Function*

Quantizing the model data on MATLAB requires a function equivalent to the `quantclip` operator found in *QuantizedSqueezeNet* that can transform each convolution kernel parameter into its quantized value.

Following the procedure detailed in section 4.4.1, we defined the following code as the `quantclip` MATLAB function which converts the real value x , to the quantized value y , with the most significant bit binary exponent msb and bit resolution $nbits$:

```
function y = quantclip(x, msb, nbits)

% Amplitude
amplitude = 2 ^ msb;

% Clip
x = max(min(x, amplitude - 1e-6), -amplitude);

% Quantization resolution (for 2's complement)
q = (2 * amplitude / 2 ^ nbits);

% Compute output
y = q * floor(x / q);

end
```

Code 14: MATLAB function that quantizes a real value with the MSB and resolution parameters specified.

5.3.2 Kernel Clipping

To calculate the MSB required to fit the values specified in the percentile limits shown on Table 2 we can use the following equation:

$$MSB = 2^{\lceil \log_2(\max(\text{abs}(\text{limits}))) \rceil}$$

Equation 11: MSB value calculated from the percentile limits.

This equation computes the minimum MSB that can represent all the values in the maximum absolute percentile limit. This equation is designed for fixed-comma binary representations in 2's complement for bipolar values.

Using the `quantclip` function with the resolution set to 32-bit, we clipped the model data with the MSB configuration calculated from the percentile limits. Evaluating those four models we obtained the following results:

Percentile	Limits		MSB	Clipping	Accuracy
100%	None		float-32	None	51%
99.99 %	-0.7315	0.6775	2^0	± 1	49.1%
99.9 %	-0.4321	0.4153	2^{-1}	± 0.5	16.4%
98 %	-0.2158	0.2088	2^{-2}	± 0.25	0.17%
95 %	-0.1693	0.1659			

Table 4: Accuracy of the model with convolution weights clipped with different values.

Using a unique MSB configuration for every layer of the model requires to select the more stringent percentile limit studied to achieve an accuracy similar to the base one. Using other configurations drastically reduces its accuracy which emphasizes the importance of choosing precise quantization parameters while adapting the model to a low-resolution environment.

5.3.3 Kernel Resolution

Following a similar strategy, we analyzed the effects on the accuracy the model when reducing the resolution of the kernel parameters. Using the 2^0 MSB parameter we generated three new models with the kernel parameters resolution reduced to 16, 8 and 4 bits. The evaluation of these models produced the following results:

Quantization	Clipping	MSB	LSB	Accuracy
<i>None</i>	<i>None</i>	<i>float-32</i>		51%
16 bits	± 1	2^0	2^{-15}	48.2%
8 bits	± 1	2^0	2^{-7}	7.16%
4 bits	± 1	2^0	2^{-3}	0.14%

Table 5: Accuracy of the model with different quantization resolutions.

These results clearly define a limit in the scope of the quantization strategy used, since it seems that any significant reduction in the parameters resolution under the 16-bit limit dramatically decreases the accuracy of the model. As the objective of this work is to achieve a high accuracy model with the lowest resolution convolutions required the use of more advanced techniques seems inevitable.

5.4 ON-MODEL QUANTIZATION

To fully understand the effects of a complete quantized convolution on the model it is required to also quantize the input and output of the convolutions. Off-model techniques do not support fully quantizing the convolution operations, so we implemented the on-model quantization scheme found in QuantizedSqueezeNet that was detailed in section 4.5.

5.4.1 Kernel Quantization

Before fully enabling the quantization of the model we checked that both the off-model and on-model strategies generated the same effects when quantizing the kernel parameters with the same configuration.

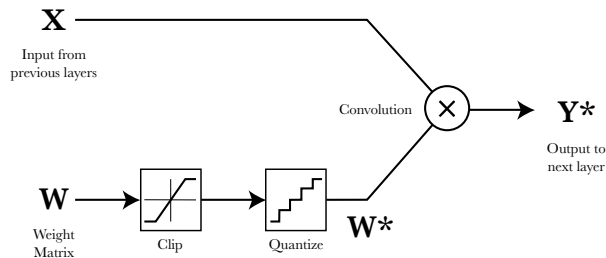


Figure 19: Structure of a convolution with quantization only applied to the weight matrix.

We configured QuantizedSqueezeNet with the quantization parameters used in the off-model cases from Table 5. For each case, we achieved to following results:

Resolution	Clipping	Accuracy	
		Off-model	On-model
<i>None</i>	<i>None</i>	<i>51%</i>	
16 bits	± 1	48.2%	49.3%
8 bits	± 1	7.16%	7.3%
4 bits	± 1	0.14%	0.11%

Table 6: Comparison on the accuracy of the model for the external and on-model quantization.

As expected, the results of the off-model and on-model strategies generated similar results as the `quantclip` tensor operator and MATLAB function produce equivalent quantizations.

5.4.2 Kernel + Input Quantization

The next step was to enable the quantization on the input branch of the convolution. In this first approach, we will setup the quantization with the same parameters for all the model layers as done with the kernel input.

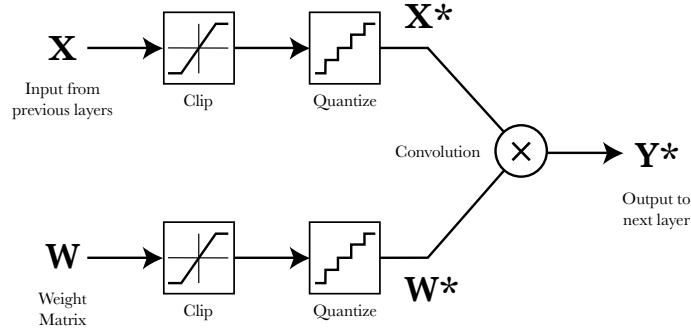


Figure 20: Structure of a convolutional layer with full quantization (ignoring the biases).

After computing the MSB quantization configurations from the percentile limits obtained from the output extraction analysis in section 5.2.2, we tried these clipping limits with a 16-bit resolution on the input while maintaining the 16-bit, 2^0 MSB on the kernel quantization:

Percentile	Limits		MSB	Clipping	Accuracy
<i>100%</i>	<i>None</i>		<i>float-32</i>	<i>None</i>	<i>51%</i>
99.99 %	-1008	571.9	2^{10}	± 1024	48.45%
99.9 %	-507	377.8	2^9	± 512	48.9%

98%	-170	184.5	2^8	± 256	45.4%
95 %	-115.8	127.3	2^7	± 128	25.7%

Table 7: Accuracy of the model with convolution inputs clipped with different values.

In this case, the most stringent percentile is not the best option, as it reduces the resolution which seems to be the determinant effect. The ± 512 clipping boundaries seem to produce the best result balancing the trade-off between underflow (too low resolution) and overflow (too small boundaries).

Using the 2^9 MSB parameter we tried other resolution values to check its effect on the convolutions input:

Resolution	Clipping	MSB	Accuracy
32 bits	None	float-32	51%
32 bits	± 512	2^9	48.08%
16 bits	± 512	2^9	48.9%
8 bits	± 512	2^9	44.52%

Table 8: Comparison on the accuracy of the model for different quantization resolutions on the input.

The reduction in the model accuracy due to the low-resolution in the input seems to be a lot less intense than in the kernel, as seen in the Table 5.

5.4.3 Full Quantization

The full quantization of the convolution requires that both the inputs and output of the operator must be quantized.

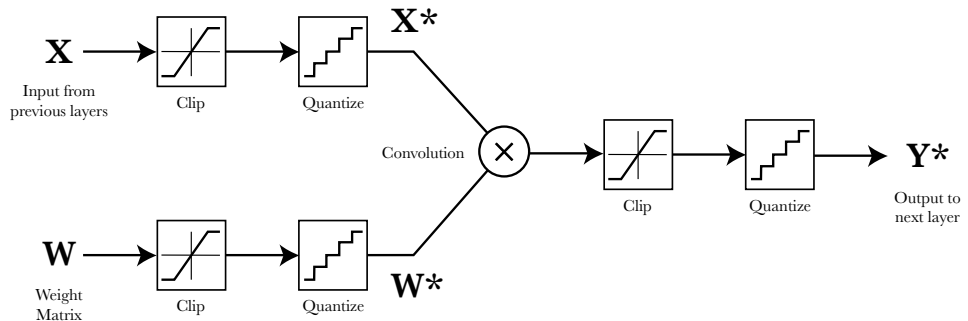


Figure 21: Structure of a convolutional layer with full quantization (ignoring the biases).

Using the 2^0 MSB for the kernel parameter and 2^9 for the input and output we setup the full quantization implementation of `QuantizedSqueezeNet` and tried various resolution configurations. We obtained the following results, being w_n the kernel bits, i_n the input bits and o_n the output bits:

Resolution			Accuracy
w_n	i_n	o_n	
8	6	6	0.08%
8	8	8	0.25%
16	8	8	43.9%
16	16	16	47.59%

Table 9: Parameters and accuracy of the full on-model quantization with various resolutions.

These results reinforce the impression that the kernel resolution is the most vulnerable value to a low-resolution environment. It also shows that static quantization is not a feasible approach to our main objective of reducing the model resolution under 16 bits while maintaining a high accuracy. To solve this inconveniency, we have designed some dynamic quantization techniques described on the next section.

5.5 DYNAMIC QUANTIZATION

5.5.1 Introduction

A dynamic or progressive quantization is a strategy to adapt a full-resolution model to a quantized state gradually during a fine-tuning training. There are multiple approaches for dynamic quantization, but typically the model starts in a full-resolution state and gradually enables the quantization of each layer. The following list enumerates some of the possible techniques:

- **Gradual Resolution Loss:** Gradually reduce the resolution of the convolutions so the model can adapt to the low-resolution state.
- **Quantize-and-Lock:** Retrain the model quantizing and locking each layer after some training epochs starting by the input layer until the output one; thus, progressively quantizing the model.
- **Quantize-while-Train:** Similar to the previous method, but the quantized layers remain in a training state until the process finishes.

The next sections detail the results achieved using the last two methods.

5.5.2 Quantize-and-Lock

Starting with the model obtained in the full initial training session described in section 5.1.2, we applied the Quantize-and-Lock strategy with four different configurations:

- **8-6-6 @ 10:** Kernel quantized with 8-bit resolution and input and output of the convolutions with 6-bit. After each layer is quantized and locked we will train the model

during 10 epochs.

- **8-6-6 @ 100**: Same configuration as before but with 100 epochs per layer, 1000 training epochs in total.
- **8-8-8 @ 10**: Same 8-bit resolution for weights, input and output of the convolutions and 10 epochs of training after each is layer quantized a locked, 100 epochs in total.
- **8-8-8 @ 100**: Same configuration as before but with 100 epochs per layer, 1000 training epochs in total.

For all the configurations, we set the same MSB parameters for every layer: 2^0 for the kernel and 2^9 for the input and output. The base learning rate was set up at 0.0004 to ensure a slow fine-tuning style quantization.

The following plot shows the evolution of the accuracy during the dynamic quantization for the four configurations.

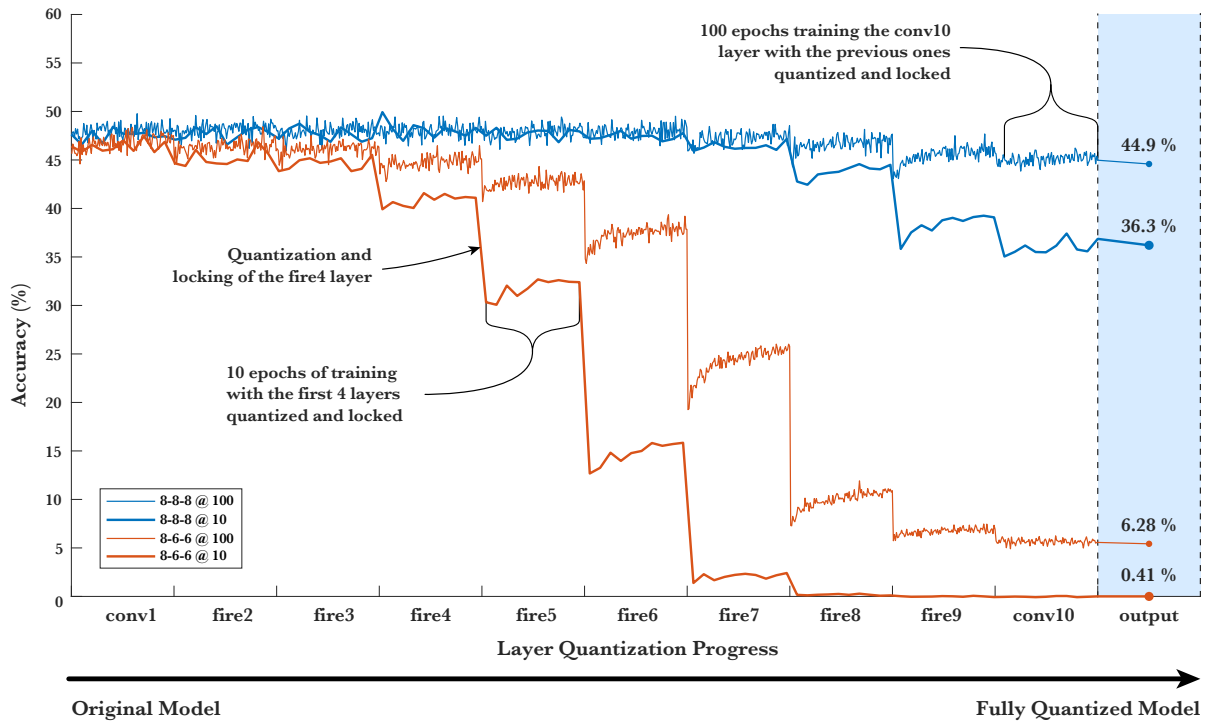


Figure 22: Evolution of the accuracy of the model with the different settings and training period during a dynamic quantization process.

This plot clearly shows the effect produced on the model after each layer is quantized. For both the 8-6-6-bit configurations the model reduces its accuracy by 40 percentage points after the quantization of four inner layers (fire4, fire5, fire6 and fire7).

It is also worth mentioning, that the accuracy recovery produced after each layer quantization due to the fine-tuning of the non-quantized layers is not very effective. For example, in the 8-8-6-bit case with 100 training epochs per layer after the quantization and locking of the fire6 layer the accuracy is reduced from 37% to 19%, a decrease of 19 points, while the recovery during the following 100 epochs is of only 6 points, up to 25%.

We can compare the acquired accuracy after the progressive quantization with the accuracy from the static quantization strategy tried before.

Resolution			Epochs per layer	Training Time	Accuracy	
w_n	i_n	o_n			Static Quantization	Dynamic Quantization
8	6	6	10	4 h	0.08%	0.41%
8	6	6	100	41 h	0.08%	6.28%
8	8	8	10	4 h	0.25%	36.3%
8	8	8	100	40 h	0.25%	44.9%

Table 10: Comparison between the accuracy of the model for different parameters during the progressive quantization vs. static quantization.

The results for the cases with 100 epochs of training per layer show a 44-point and a 6-point increase in the accuracy of the model after the dynamic quantization process in comparison with the static quantization result.

These enhancements show that is possible to improve and adapt our existing model to a new more constrained state using dynamic quantization techniques.

5.5.3 Quantize-and-Lock + Custom MSB

We can further improve the accuracy of the model by implementing custom per-layer MSB configuration. As already commented in sections 5.3.2 and 5.4.2, the best MSB configurations for each layer can be computed using the Equation 11.

The following table shows the MSB binary exponent computed for the kernel and input/output quantization parameters on each layer and percentile. For example, the conv1 layer quantization should be configured with a 2^1 MSB for every percentile case.

Layer	Kernel				Input and Output			
	95%	98%	99.9%	99.99%	95%	98%	99.9%	99.99%
conv1	1	1	1	1	7	8	8	8
fire2	0	0	1	1	7	8	9	9
fire3	0	0	0	0	9	10	11	11
fire4	0	0	0	0	8	8	9	9
fire5	0	0	0	0	8	9	9	10

fire6	0	0	0	-1	8	8	9	9
fire7	0	0	0	-1	8	8	9	9
fire8	0	0	0	0	8	8	9	9
fire9	0	0	0	-1	7	7	8	9
conv10	0	0	0	-2	7	7	8	9

Table 11: Best MSB configurations for each layer of the model.

Using these values, we configured the **QuantizedSqueezeNet** model with the 99.9%, 98% and 95% percentile MSB parameters. To compare the effect of each percentile case, we applied the Quantize-and-Lock progressive quantization technique to every configuration with the same convolution resolutions: 8 bits for the kernel and 6 bits for both the input and output. In all cases the initial model was the one obtained during the full initial training session described in section 5.1.2 and the system will apply 10 training epochs to the non-quantized layers after each quantization.

The following plot shows the evolution during the progressive quantization of each one of the configurations in comparison with the 8-6-6-bit with non-custom MSB per layer model from the last section.

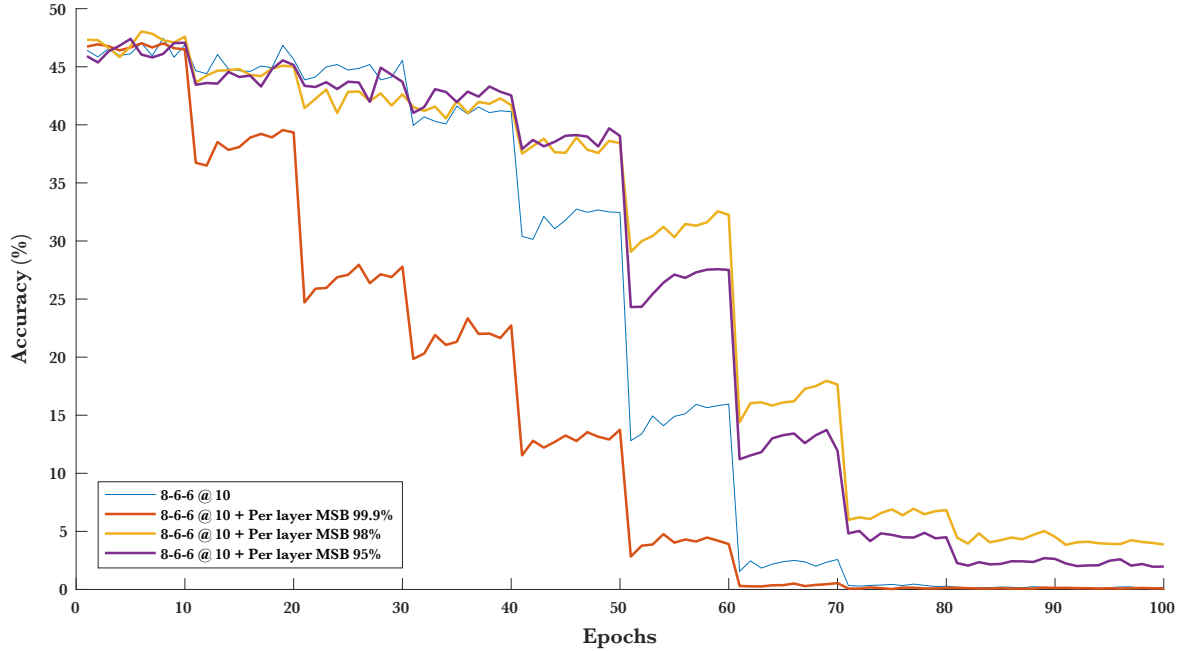


Figure 23: Dynamic quantization of the model with 8-6-6-bit resolution convolutions with three custom-per-layer MSB settings in comparison with the model with generic MSB for all layers.

We can convey from the results shown in this plot that applying custom MSB parameters in each layer of the model can increase the accuracy obtained during a progressive quantization process. Both the parameters computed from the 95% as the 98% percentile produced better results than the generic MSB configuration for all layers. The 99.9% percentile parameters scored worse than

the generic approach, probably due to an underflow effect (too low resolution for the model to distinguish between similar values).

To extract better results with these configurations we proceeded to run a long dynamic quantization with the MSB parameters obtained from the 98% percentile boundaries. We setup to models with 8-8-8-bit and 8-6-6-bit resolution configurations and 100 training epochs after each quantization. The following table exposes the results in comparison with the results obtained with the generic MSB approach from Table 10:

Resolution			Epochs per layer	Training Time	Accuracy	
w_n	i_n	o_n			Generic MSB	Custom MSB
8	6	6	100	43 h	6.28%	30.31%
8	8	8	100	44 h	44.9%	45.6%

Table 12: Accuracy of the model for different resolutions comparing the generic MSB and custom MSB approach.

These results suggest that the 8-8-8-bit model had already resolution enough to achieve the saturation accuracy while the 8-6-6-bit configuration had a big space for improvement. The custom MSB 8-6-6-bit model achieves a tolerable accuracy and evidences that a generic MSB approach is not optimal when reducing the model resolution to very small values.

5.5.4 Quantize-while-Train

While the Quantize-and-Lock strategy produced some usable models, it also proved that training only non-quantized layers while fine-tuning implied low accuracy recovery after each layer was quantized. To solve this issue the Quantize-while-Train approach involves the quantization of all the model while quantizing each layer, starting by the input stage until the output convolution. Training quantized layers required some modifications on the system as denoted in section 4.5.1.

Using the custom MSB approach, we applied the Quantize-while-Train strategy on the model obtained during the full initial training with the following parameters:

- **8-6-6 @ 10:** Kernel quantized with 8-bit resolution and input and output of the convolutions with 6-bit. After each layer is quantized we will train the full model during 10 epochs.
- **6-6-6 @ 10:** Both kernel, input and output quantized with 8-bit. After each layer is quantized we will train the full model during 10 epochs.
- **4-4-4 @ 10:** Same as the previous configuration but with 4-bit resolution.

The following plot shows the evolution of the accuracy of the training during the Quantize-while-Train quantization for the four configurations.

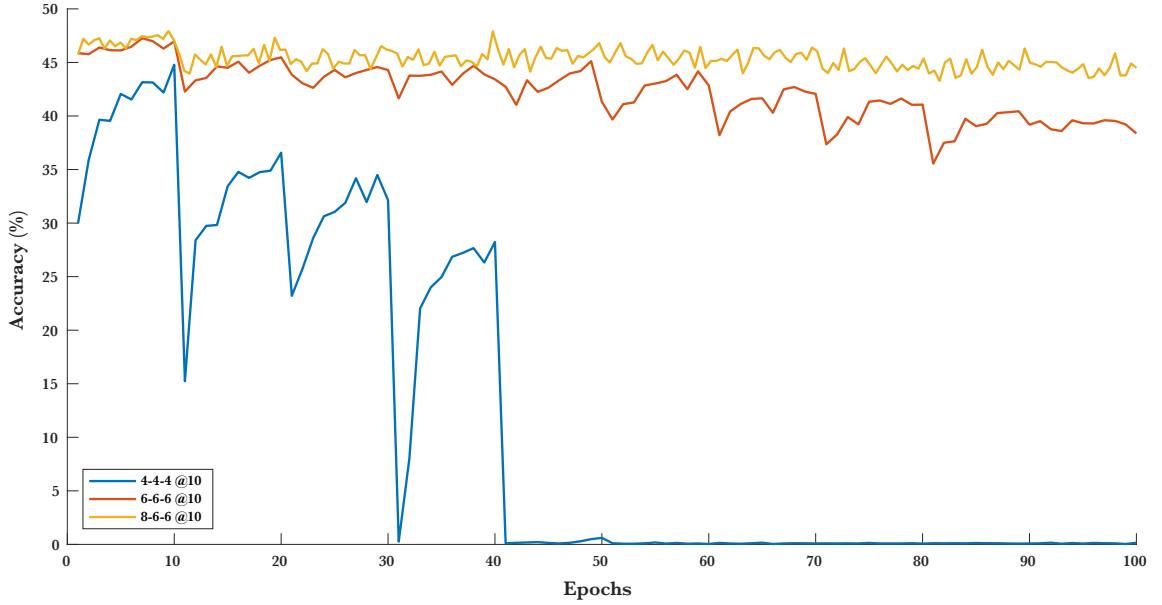


Figure 24: Evolution of the accuracy of the model with the different settings and during a Quantize-while-Train process.

From these results, we can convey that Quantize-while-Train is a more efficient technique to achieve low-resolution models with high accuracy than the previous method and it does not require long duration fine-tunings. The main advantage in comparison with Quantize-and-Lock is that it recovers the accuracy gracefully after each layer quantization. This approach was able to produce both 8-6-6-bit and 6-6-6-bit models with high accuracy with short-duration trainings: 44.5% and 38.16%. In the 4-4-4-bit case, after the system quantized the `fire5` layer (epoch 40) the model was not able to recover successfully.

The following table summarizes the results obtained in these last experiments compared to the Quantize-and-Lock + Custom MSB method.

Resolution			Epochs per layer	Accuracy	
w_n	i_n	o_n		Quantize-and-Lock	Quantize-while-Train
8	6	6	10	3.97%	44.5 %
6	6	6	10	0.15%	37.16%
4	4	4	10	0.1%	0.14%

Table 13: Comparison on the model accuracy obtained with the Quantize-and-Lock vs. Quantize-while-Train.

5.5.5 Long Duration Quantize-while-Train

The concluding collection of trainings we executed was an extended version of the previous experiment. We setup the following configurations and applied the Quantize-while-Train technique:

- **6-6-6 @ 100:** Both kernel, input and output quantized with 6 bits. After each layer is quantized we will train the full model during 100 epochs.
- **4-4-4 @ 100:** Same setup as the previous training but with 4-bit resolution on both the kernel, input and output.

The following graph shows the evolution of the long-duration Quantize-while-Train process for each configuration.

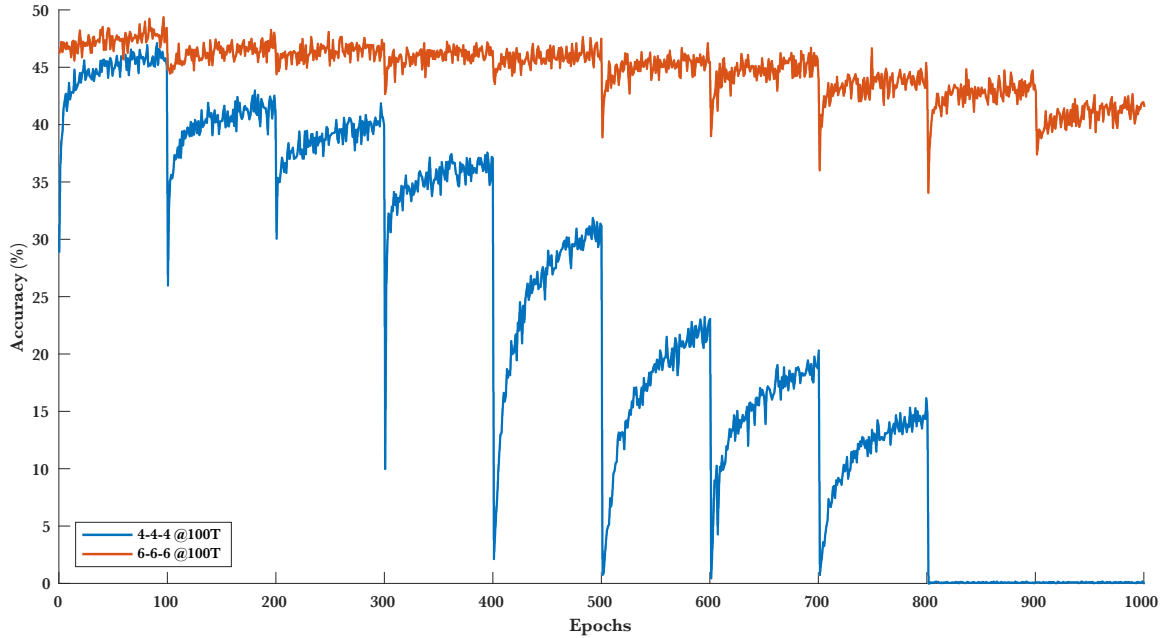


Figure 25: Evolution of the model accuracy during a long Quantize-while-Train process.

The following table summarizes the results of this last long-duration trainings.

Resolution			Epochs per layer	Accuracy
w_n	i_n	o_n		
6	6	6	100	41.30%
4	4	4	100	0.19%

Table 14: Model accuracy obtained for each configuration with the long-duration Quantize-while-Train technique.

Like during the short-duration Quantize-while-Train results, the 6-6-6-bit model achieved a high accuracy during the quantization, 41.30%, which represents an 81% of the base full-resolution result. We must remark that a 6-bit model would require only 18.75% of the parameters store space of the standard 32-bit model and would have an equivalent lower power impact in platforms with hardware support for low-resolution convolution operations like the PULP platform. This data exemplifies that applying dynamic quantization techniques to a fully trained model can produce near full-resolution accuracies on low-resolution models.

The 4-4-4-bit model evolution shows that even with a 10-times longer training, the model was not able to recover the accuracy losses produced during the quantization of the `fire9` layer (epoch 800). Solving this problem would require either longer trainings, a custom-tuned optimizer or more advanced dynamic quantization techniques.

6 CONCLUSIONS AND FUTURE DEVELOPMENT

6.1 CONCLUSIONS

Along this work we have explore the requirements needed and options available to adapt a convolutional neural network model to a low-resolution environment. This procedure requires the design of a software architecture, data analysis and the execution of advanced Deep Learning techniques.

First, we introduced the basic notion of neural networks, its evolution and structure, its implementation on Deep Learning software and common topologies used for image classification. We also detailed the training process of convolutional neural network and how the optimizer flow works.

Next, we described the NeuralNet software architecture we designed to construct multiple neural network models and evaluate, train or analyze them. We explored the different pieces that conform the system and how it was implemented to allow various models and parameter testing.

To deeply understand the SqueezeNet model we described its topology and the reimplementaion we did atop of the NeuralNet system. We detailed the modifications required to adapt a high-resolution model to a low-resolution environment with on-model quantization techniques.

Finally, we exhaustively explored the multiple experimental results obtained from the various quantization techniques implemented and tested. From all these results, we can highlight that the best option available to achieve a model immune to low-resolution environments is the Quantize-while-Train procedure due to its ability to recover the accuracy losses produced during the quantization of each layer. Also, it is remarkable that customizing the most significant bits of each model layer can report important accuracy increases during the process. Additionally, we have seen a positive effect of longer trainings during a dynamic quantization process, up to 100 epochs per quantization, without any over-fitting effect.

6.2 FUTURE DEVELOPMENT

The main two lines of future development that this work opens are the following:

- Complete the implementation of the NeuralNet system in a low-power platform provided with a Hardware Convolution Engine like the PULP platform and check the power efficiency of the model along its accuracy.
- Enhance the dynamic quantization techniques explained and try other configurations. This could involve other sequential evolution, for example, starting the progressive quantization by the last layers or in a random order; also, implementing the Gradual Resolution Loss strategy mentioned in chapter 5.

Other options could be to apply the techniques described to other convolutional neural network topologies or to port the system to other Deep Learning software.

7 BIBLIOGRAPHY

- Abadi, M. (2015). *TensorFlow*. Retrieved from <https://www.tensorflow.org>
- Apple Inc. (2017). *Core ML: Integrate machine learning models into your app*. Retrieved from Apple Developer Documentation: <https://developer.apple.com/documentation/coreml>
- Bergstra, J. (2010). *Theano*. Retrieved from <http://www.deeplearning.net/software/theano/>
- Chollet, F. (2015). *Keras*. Retrieved from <https://keras.io>
- Collobert, R. (2011). *Torch*. Retrieved from <http://torch.ch>
- Conti, F., & Benini, L. (2015). *A Ultra-Low-Energy Convolution Engine for Fast Brain-Inspired Vision in Multicore Clusters*.
- Courbariaux, M., Hubara, I., Soudry, D., El-Yaniv, R., & Bengio, Y. (2016). Binarized Neural Networks: Training Neural Networks with Weights and Activations Constrained to +1 or -1. *arXiv*.
- DeepScale. (2016). *DeepScale/SqueezeNet*. Retrieved from GitHub: <https://github.com/DeepScale/SqueezeNet>
- He, K., Zhang, X., Ren, S., & Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *ICCV*.
- Iandola, F., Moskewicz, M., Ashraf, K., Han, S., & Keutzer, K. (2016). SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size. *CoRR, abs/1602.07360*.
- ImageNet. (2012). *Large Scale Visual Recognition Challenge 2012 (ILSVRC2012)*. Retrieved from ImageNet: <http://www.image-net.org/challenges/LSVRC/2012/results.html>
- Jia, Y. (2013). *Caffe*. Retrieved from <http://caffe.berkeleyvision.org>
- Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet Classification with Deep Convolutional Neural Networks. *NIPS 2012*.
- LeCun, Bottou, Bengio, & Haffner. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*.
- Simonyan, K., & Zisserman, A. (2015). *Very Deep Convolutional Networks for Large-Scale Image Recognition*. Retrieved from <https://arxiv.org/pdf/1409.1556.pdf>
- Szegedy, C., Liu, W., & Jia, Y. (2014). *Going Deeper with Convolutions*. Retrieved from <https://arxiv.org/abs/1409.4842>